Exploiting Rich Content

Riley Hassell

As rich Internet application (RIA) technologies flourish in the marketplace security professionals begun to wonder what impact RIA will have on security landscape. I decided to perform an assessment of one of the most widely deployed technologies, Adobe Flash, and in the process discovered several issues that could be used to compromise systems with Adobe Flash installed. During the audit a large group of issues was uncovered that initially appeared harmless. Later in this paper I will describe how attackers can exploit these perceived low risk issues to have a much deeper impact.

- Flash Overview
- Testing Methodology
- Manual Review
- Reverse Engineering
- Automated Testing
- Test Results

Flash Overview

Adobe Flash is one of the most widely deployed software technologies to date. The (Millward Brown) estimates 99% of internet-connected computers are Flash enabled. Flash delivers a wide variety of rich multimedia feature to its clients including: rich web based application, video streaming, gaming, and

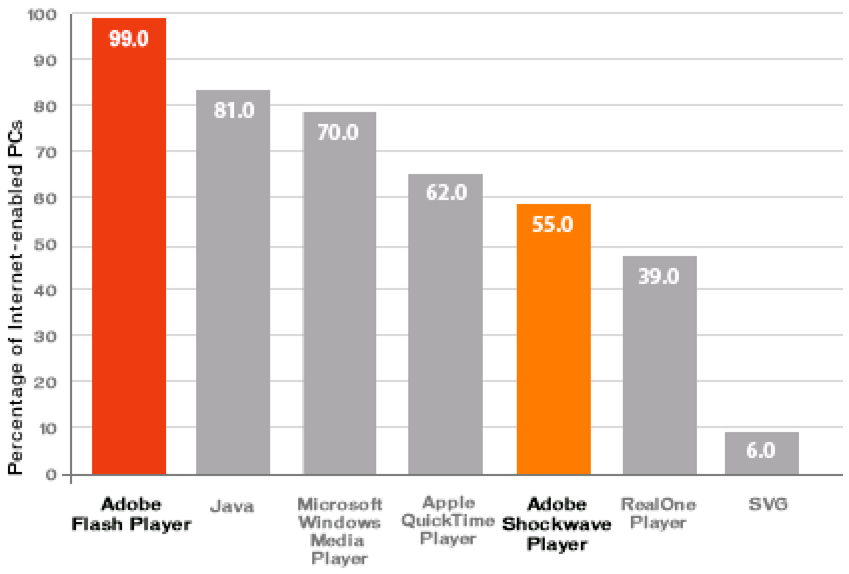
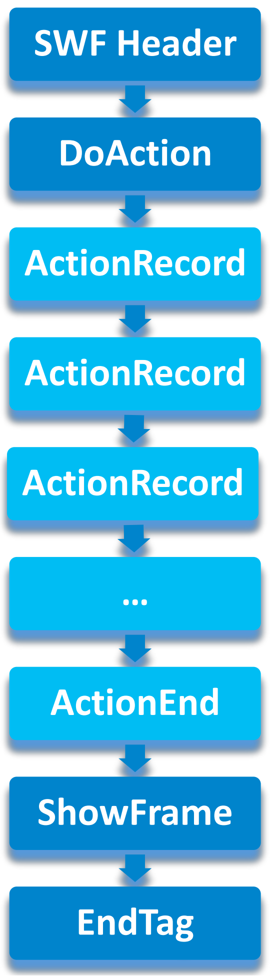


Figure 1 Flash per Internet enabled desktops. [1]

The SWF version 9[2] file format consists of 64 tag types that are parsed sequentially through the file. Tags have a TLV (type, length, value) structure. Some tags contain embedded data such as bitmaps, sounds, fonts, and even video. Several tags support sub-types of various depths. One such tag is DoAction. This tag contains compiled ActionScript 2.0 byte codes.

[1] Millward Brown survey: http://www.adobe.com/products/player_census/flashplayer/

[2]http://www.adobe.com/devnet/swf/pdf/swf_file_format_spec_v9.pdf



Many features are exposed through tags and tag data. One of the more powerful features exposed is ActionScript. Designed initially for simple animation but has since been extended to offer rich functionality. ActionScript is based on the ECMAScript standard therefore it is very similar to JavaScript. ActionScript is supported by all popular Flash players. ActionScript 2.0 when compiled is converted to ActionRecord sub tags. The ActionRecord sub tags are stored within DoAction tag data. A stream of ActionRecord(s) is terminated with record type of ActionEnd.

Testing methodology

Based on current research and public statistics we know the following:

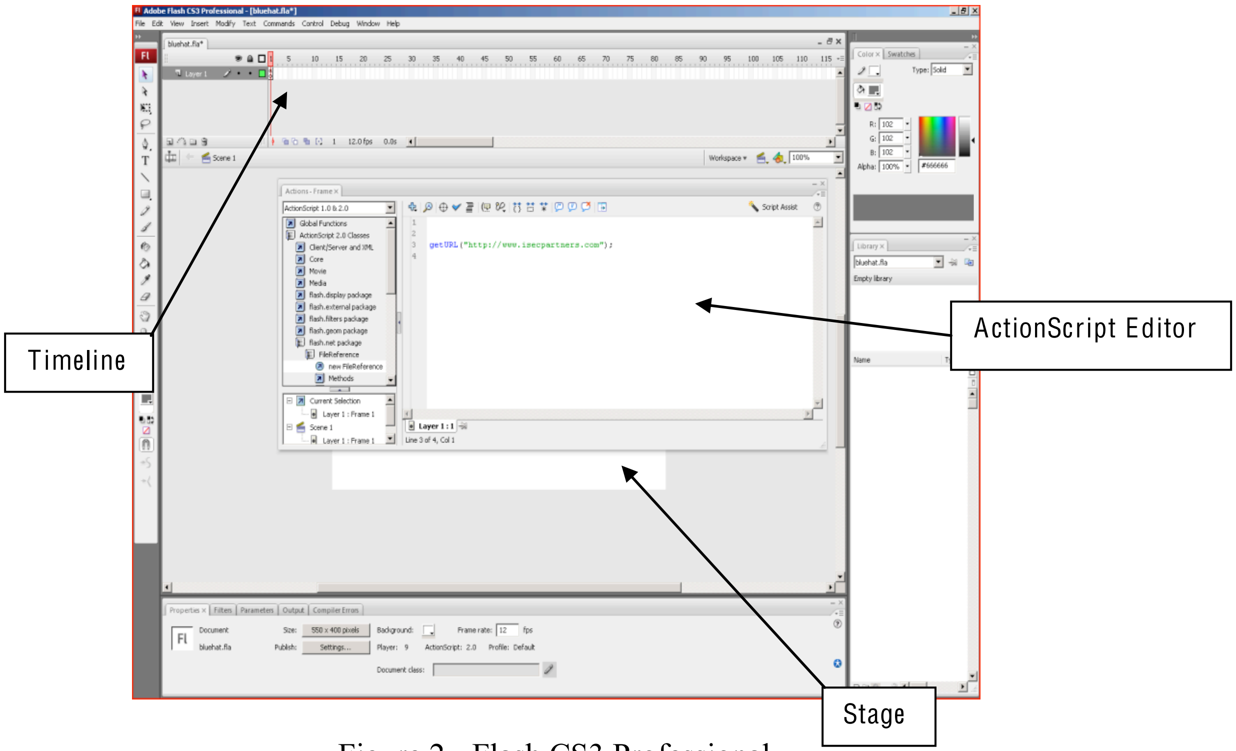
- Flash is available on all major desktop operating systems
- Flash Player is often installed default by vendors
- When Player not installed default by System vendor, user will usually install the Flash player/
- ActionScript 2 (AS2) is supported by all popular players. Even FlashLite, the Flash player for mobile device supports ActionScript 2.0

The version penetration portion of the (Millward Brown) survey indicates that on average 99% of systems in mature markets support Action Script 2.0. Adobe indicated in their (PSIRT advisory) indicated that Flash Player 9.0.124.0 and earlier were affected by my findings. Therefore the issues discovered in the core AS2 runtime could potentially affect 99% of Internet-enabled computers. Due to its healthy install base, and the fact that it typically runs automatically when pages contain flash content, exploitable problems with Flash implementations would represent a potentially major risk to Internet users.. Software with large installed bases may be prioritized by hackers and worm writers. One goal of this project was to investigate the existence of vulnerabilities that could be exploited across the various Flash-capable OSs. Results were provided to Adobe for remediation. Testing was broken up into several phases.

http://www.adobe.com/products/player_census/flashplayer/version_penetration.html

Manual Testing

Adobe's Flash Professional is the most popular development environment for Flash applications. Using Flash Professional I created simple Flash movies with ActionScript. I then dissected my creations and reviewed the compiled movies in depth to better understand the ActionScript runtime.



```
getURL("http://www.isecpartners.com");
```

Figure 2 - Flash CS3 Professional

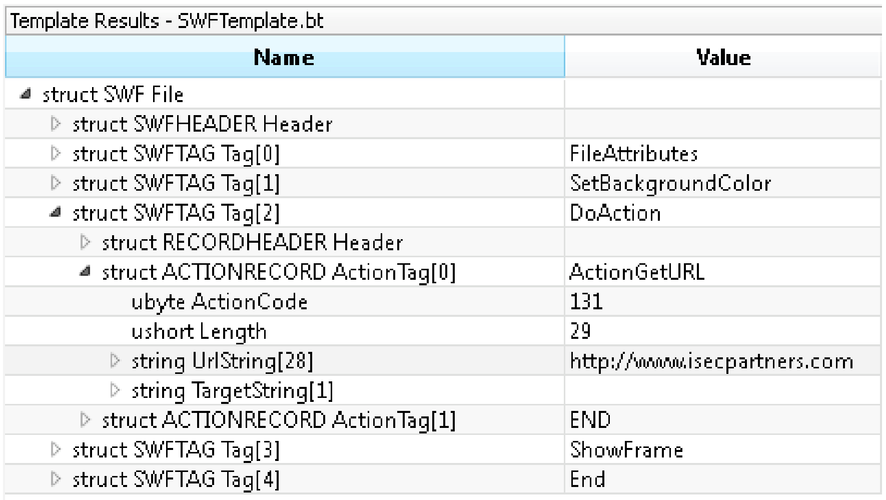
The `stage` is the large white rectangle where the movie frame design is conducted. The user can attach various multimedia assets and actions to each frame.

The `timeline` is a linear representation of the frames within a movie. Flash developers typically manage frames and frame contents using the timeline.

The `ActionScript Editor` control is used for editing and managing ActionScript attached to the current Flash movie. In our screen shot shown above the following ActionScript source has been added to the first frame:

```
getURL("http://www.isecpartners.com");
```

As you may have already guessed when this line of ActionScript is executed by the player a browser control will be created and redirected to the URL provided (e.g. http://www.isecpartners.com). The SweetScape 010 Editor[3] now includes a file format template applied to a SWF movie a breakdown of the file contents will be provided in a tree view that the user can navigate. The following screenshot depicts the output returned once the SWF template was applied to our sample SWF movie that makes use of *getURL*.


Template Results - SWFTemplate.bt

Name	Value
struct SWF File	
struct SWFHEADER Header	
struct SWFTAG Tag[0]	FileAttributes
struct SWFTAG Tag[1]	SetBackgroundColor
struct SWFTAG Tag[2]	DoAction
struct RECORDHEADER Header	
struct ACTIONRECORD ActionTag[0]	ActionGetURL
ubyte ActionCode	131
ushort Length	29
string UrlString[28]	http://www.isecpartners.com
string TargetString[1]	
struct ACTIONRECORD ActionTag[1]	END
struct SWFTAG Tag[3]	ShowFrame
struct SWFTAG Tag[4]	End

Figure 3 – SWF Template for 010 Editor

[3] SweetScape 010 Editor: http://sweetscape.com/010editor/

Reverse Engineering

I performed a series of short reverse engineering sessions in order to get an idea of what was happening "under the hood" in the popular Flash players. I observed that players typically do not validate the sizes of compartmentalized data structures. Many of the features across versions appear to be grouped together in code. The existing code was not split up for each major version's support. This development style are one of the many characteristics of the flash player that allow make it so versatile yet permit

Automated Testing

An automated testing platform Fault Injection for Reverse Engineers (FIRE) was developed over the last two years to deal with many of the problems encountered when testing complex file formats. This framework when applied to Flash was termed Flash-FIRE. The FIRE framework was augmented to incorporate event hooking through process instrumentation. We use this method of handling events to drive and synchronize the delivery of faults to the target application. Event Driven Fault Injection (EDFI) offers several major gains when performing fault injection as will be reviewed later in this section.

Gather Input

The FIRE framework is a mutation based fault injection framework, meaning input is mutated or altered and then supplied to a target function, module, or application. Since input is required for testing it must be gathered. A python script was developed that uses the Google SOAP API to find SWF movies and then downloads and indexes them by unique MD5 hash.

Survey Input

Gathered input is surveyed for interesting regions and offsets the regions are saved into a list of vectors. The surveying logic differs between target technologies. Survey logic Flash will skip large textual data regions such XML, HTML and ASCII. Regions that contain binary data such as ActionScript regions will be marked for fault injection tested in the next phase.

Process Instrumentation

Now that input has been prepared the test harness must be initialized and attached to the target technology, in this case a browser application with Flash Player loaded as a COM object. When FlashFIRE starts it will invoke Internet Explorer with a default homepage set to simple HTML page with a basic SWF movie. Internet Explorer will be invoked and monitored by the FlashFIRE debugger. This debugger will monitor the Internet Explorer instance and detect perform actions based on a wide variety of events. One such event is the loading of the Flash Player module. When this occurs a breakpoint will be inserted inside the Flash Player at a *CreateWindow* call. This code point is executed after a Flash movie has been process and right before the visual aspects are painted onto the screen. By monitoring the execution of this point and a few other error conditions points the state of the fault injection can be closely measured.

Mutate Input

Batches of files are retrieved from the catalog and for each iteration of testing a file is pulled from the batch and mutated. The file is mutated by injecting a variety of faulty input, e.g. for integer overflows 8bit,16bit and 32bit integer fields that trigger common integer overflow vulnerabilities are injected into surveyed regions. Once the fault has been injected an event is sent to target application to trigger it to load the test input (SWF file).

Process Monitoring

During the instrumentation phase breakpoints were set on several key code execution control paths in the target application. Each time one of these code points is executed a breakpoint will be hit and a corresponding event will be generated with FlashFIRE and delivered to the necessary listener.

Module Load Event

This event is fired when a selected module has been loaded in the target address space.

Fault Delivered Event

This event is raised when the fault input has been completely processed.

Application Failure Event

This event is raised when recoverable errors such as handled exceptions are encountered.

Application Critical Failure Event

Monitor points are setup in the target application in code paths that are only exercised when critical failures are encountered. These failures included: failure stack/heap cookie checks, exit process, and unhandled exceptions.

Post Mortem bug Analysis (PMA)

When an Application Failure Event or Application Critical Failure Event is encountered the FlashFIRE debugger will record the case by collecting the current input stream (with the injected fault), the thread context, and the stack trace from the current thread. These items will be saved into a bug catalog of current findings.

Bug Cache

Another important enhancement to the FlashFIRE debugger is exception caching. When an application error is encountered the call stack is validated to be not null and at least three frames in depth. If this criterion is met a hash is created from the call stack and current EIP. This hash is checked against a list of previously encountered hash. If the hash does not exist it is created and the bug is cataloged. If it already exists then no further processing of the bug occurs.

Test Results

In this section the results of testing effort will be discussed and reviewed in depth where pertinent. The follow bullet points summarize the testing effort and results:

- 3 million injections in 36 hours of testing
- 23 unique vulnerabilities identified
- 785 unique paths to vulnerable code sequences identified

Targeted testing was applied to the ActionScript 2 virtual machine used by the Adobe Flash player. Several issues were identified which could lead to denial of service, information disclosure or code execution when parsing a malicious SWF file. The majority of testing occurred during 120 hours of automated SWF-specific fault injection testing in which several hundred unique control paths were identified that trigger bugs and/or potential vulnerabilities in the Adobe Flash Player. Paths leading to duplicate issues where condensed down to a number of unique problems in the Adobe Flash Player. The primary cause for these vulnerabilities appears to be simple failures in verifying the bounds of compartmentalized data structures.

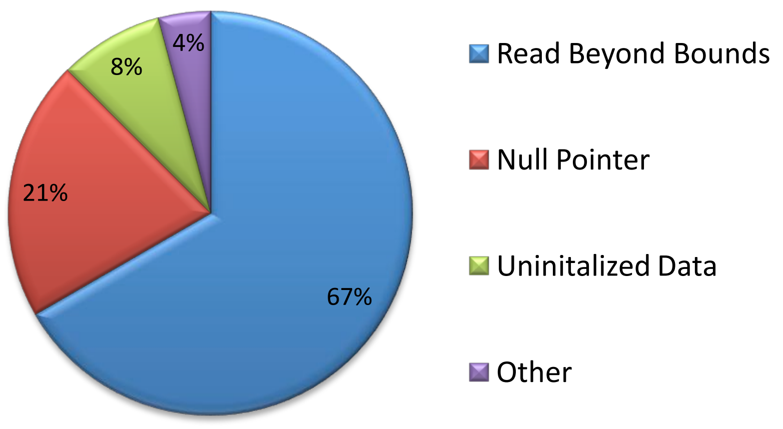


Figure 4 – Automated Testing Results

As shown in the graph above the majority of the issues discovered are out of bounds read operations. These are typically caught by operating system exception handling and converted into an error.

Read beyond bounds

A read beyond bounds occurs when a piece of code reads beyond the bounds of the data element it is intended to read. This type of bug is very common in code that deals with complex binary structures.

SWF for example is a flat structure (file format) that consists of tags and these tag objects can have similar tag objects embedded within them. When this tag chain is traversed the length fields are used when retrieving content from the SWF into the player memory. For example the following code would perform a read beyond bounds if the length within the block structure (blk -> len) is greater than the actual size of the data element:

```
typedef struct block {
    int len;
    void *data;
} BLOCK;

char *url = NULL;

int GetElement(BLOCK *blk) {

      if(blk->len > 2048)
      {
            printf("Invalid block size!\n");
            return -1;
      }
```

```
        url = (char *)calloc(blk->len,1);

        memcpy(url, blk->data, blk->len);

        return 0;
}
```

This small example demonstrates one of the principles behind why read beyond issues are so prevalent: Dynamically sized data elements are very difficult to measure in size and the lengths supplied with those fields are usually trusted. With copy operations the size of the destination buffer is usually known and a write beyond bounds (buffer/heap overflow) can be prevented by verifying that the size of the source buffer is not greater than the size of the destination buffer. On the other side of the transaction not much is known.

Example read beyond bounds in Flash

In the case of the *DefineConstantPool* record we were able supply an arbitrary constant count. The player then parses constant values (strings) from the string table, and continues reading null terminated strings in the adjacent tag data, eventually reading from memory adjacent to the Flash movie. References to these values are stored in a table of constants that can be later accessed using a set of action records. A proof of concept was developed and presented to the vendor to demonstrate the threat of read beyond bounds issues to complex file formats such as the SWF file format.

The proof of concept SWF movie will 255 strings from adjacent heap memory, create a text file on the stage and write the contents of the strings to the stage. The following is pseudo code for the proof of concept SWF that was created to demonstrate the exploitability of a read beyond bound issue.

```
var heapstr = new Array();

heapstr[0] = "I";
heapstr[1] = "S";
heapstr[2] = "E";
heapstr[3] = "C";
heapstr[4] = const_pool_string_from_heap[0]
heapstr[5] = const_pool_string_from_heap[1]
heapstr[6] = const_pool_string_from_heap[2]
...
heapstr[259] = const_pool_string_from_heap[255]

var buffer:String = "";

for(var i = 0;i < 259;i++)
```

```
        buffer += heapstr[i];

    createTextField("tf", 1, 10, 10, 400, 100);
    tf.text = buffer;
```

When the POC is loaded into the browser a small subset of the heap strings will be written to a textbox on the browser page. For example when the author tested the proof of concept on his vulnerable test system the following was displayed in the browser:

```
IheapstrArrayISECbufferitfcreateTextFieldmultiline-
wordWraptext?————————————————™?
text?————————————————™?
————————————————@ffer
Lž file:///C:/Documents%20and%20Settings/consultant/Des
ktop/gen.swfction|
„€ H$0————————————————‚p�p�
```

Loading this demonstration SWF movie in different popular browsers yielded different outcomes due to the memory layout of the browsers at the time the movie was loaded. Often the content was sensitive, such as the username of the current logged in user, or local path to the loaded content. The exploit discussed only retrieves 255 strings from the adjacent heap. A maximum of 65535 (size of constant pool) strings could be retrieved, permitting an attacker to retrieve large portions of browser heap memory.

Storage and retrieval

An attacker must accomplish two things to exploit most read beyond bounds issue. First the data that is read from memory must be stored somewhere more easily accessible to the attacker temporarily. After the data has been stored it must be delivered to the attacker. In the exploitation of the *DefineConstantPool* vulnerability the storage phase occurs when the strings are read from the heap and stored into the constant pool string table. The retrieval would occur then the pool entries were concatenated into a buffer and sent to the browser display.

Another important factor in Read Beyond Bounds (RBB) exploitation is how the application treats the data during the storage and retrieval. For example if during the storage of the data, the copy operation is performed a using a strict size (i.e. *memcpy*) then the data may contain nulls. If the data was copied using a null terminating copy function (i.e. *strncpy*) then the data copied could be significantly smaller. Ideally an attacker would want to find a storage and retrieval combination that used strictly sized memory copy operations.

Heap Grooming

This technique has been around for nearly a decade but has recently gotten attention publicly[4]. If an attacker can influence a target application to allocate memory in sizes of their choosing and retain the allocations temporarily, they can place attacker supplied data in chosen regions of the target processes heap. Heap grooming has been demonstrated as an assisting exploit method for heap overflows but the author found that it can also be used to order the heap for other exploitation methods such as read beyond bounds. For example when exploiting a read beyond bounds issue where an attacker can only read a few hundred bytes beyond the end of a heap block. By defragmenting the heap and fan attacker can move the originating read beyond bounds block throughout the heap and each new read will capture new portions of heap memory. This can be repeated until something of interested is acquired.

Same Origin Impact

The same origin policy[5] is critical security concept that when enforced correctly prevents site content from accessing the content from another site. This malicious behavior is usually attempted through scripting. Issues such as the read beyond bounds issue allow an attacker to peek into browser memory and potentially read content from other sites. This could include session cookie, usernames and password and virtually anything else in browser heap memory.

Prevalence in modern software

Software security audits frequently scan for operations that write data to a destination. Parsers, null terminated copies, and list management often contain boundary writing issues i.e. buffer overflows. The out of bound reading of data is often overlooked and not reviewed during an audit. Secondly when read boundary issues occur they often do not cause a software failure, therefore they often go unnoticed.

Conclusion

While initially this bug class appeared fairly benign the author found several interesting qualities about RBB issues. These read beyond issues can often be exploited to retrieve sensitive data from the browser process. This issue doesn't require any addressing, ma-

[4] http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf
[5] http://en.wikipedia.org/wiki/Same_origin_policy

chine code, or other system dependent characteristics. The proof of concept shown earlier has been tested: Windows 2000, Windows XP, Windows Vista, Windows 2003, Mac OS and various flavors of Linux.

Acknowledgements
Josh Zelonis for the creation of the 010 Editor SWF and assistance analyzing, testing and documenting vulnerability discoveries from automated testing portion of this project.

iSEC Partners, Inc.

Appendix A: About iSEC Partners, Inc.

iSEC Partners is a proven full-service security consulting firm, dedicated to making Software Secure. Our focus areas include:

- Mobile Application Security
- Web Application Security
- Client/Server Security
- Continuous Web Application Scanning (Automated/Manual)

Published Books

Notable Presentations

RSA CONFERENCE Black Hat

web2.0 EXPO InfoWorld COMPUTERWORLD SNIA STORAGE NETWORKING WORLD

Whitepaper, Tools, Advisories, & SDL Products

- 11 Published Whitepapers
  - Including the first whitepaper on CSRF
- 32 Free Security Tools
  - Application, Infrastructure, Mobile, VoIP, & Storage
- 8 Advisories
  - Including products from Apple, Adobe, and Lenovo
- Free SDL Products
  - SecurityQA Toolbar (Automated Web Application Testing)
  - Code Coach (Secure Code Enforcement and Scanning)
  - Computer Based Training (Java & WebApp Security)