IBM Global Technology Services
April 2008

Application-Specific Attacks: Leveraging the ActionScript Virtual Machine

By Mark Dowd
X-Force Researcher IBM Internet Security Systems
(markdowd@au1.ibm.com)

Contents

Introduction

Memory corruption vulnerabilities are becoming increasingly difficult to exploit, largely due to the protection mechanisms being integrated into most modern operating systems. As general protection mechanisms evolve, attackers are engaging in more specific, low-level application-targeted attacks. In order to refine general countermeasures (or at least raise awareness of their shortcomings), it is important to first understand how memory corruption vulnerabilities are exploited in some unique scenarios.

The following case study describes a unique exploitation scenario using a recently disclosed flash vulnerability that was reported to Adobe by IBM (advisory available at http://www.iss.net/threats/289.html). At first the vulnerability seemed to offer limited exploitation options, but further analysis uncovered an application-specific attack that results in reliable, consistent exploitation. Achieving the same exploitation with more conventional methods is unlikely. The technique presented leverages functionality provided by the ActionScript Virtual Machine – an integral part of Adobe Flash Player. Further, it will be shown that the vulnerability can be successfully exploited without leaving telltale signs, such as a browser crash following the attack.

Although this document deals specifically with the Win32/intel platform, similar attacks can most likely be carried out on the many other platforms flash is available for. In particular, some of the methodology discussed might be useful for constructing a robust exploit on Unix platforms as well as several embedded platforms. Understanding the specific scenarios used to exploit memory corruption vulnerabilities will help improve protection strategies.

The Vulnerability

Before discussing exploitation, consider the nature of the vulnerability. Flash is vulnerable to a memory corruption issue when processing a malicious Shockwave Flash (SWF) file. An SWF file is simply a data file interpreted by the Flash Player. It contains a header followed by a series of records (referred to as 'tags') which describe various elements of a flash application – including images, sprites, frames, buttons and so on. The file format is fully documented by Adobe/Macromedia at http://download.macromedia.com/pub/flash/licensing/file_format_specification_v9.pdf.

The vulnerability in question is an exploitable NULL-pointer dereference of sorts, which enables an attacker to write data to an (almost) arbitrary offset from address 0x00000000. The bug is quite simple; it occurs when processing the "DefineSceneAndFrameLabelData" tag of an SWF file (tag ID 0x56). As the name suggests, the tag defines label data for scenes and frames within a movie clip. The structure for the data in this tag is as follows:

```
// supporting structures
SceneData
{
        UI32 FrameOffset
        String SceneName
}
FrameData
{
        UI32 FrameNumber
        String FrameLabel
}

// Tag structure
DefineSceneAndFrameLabelData
{
        RecordHeader Header
        UI32 SceneCount
        SceneData Scenes[SceneCount]
        UI32 FrameCount
        FrameData Frames[FrameCount]
}
```

(The RecordHeader and UI32 data types are defined in the Flash File Specification link mentioned previously.)

The “DefineSceneAndFrameLabelData” is a variable length tag in which the “Scene Count” integer is supplied, followed by the number of scene records specified by the Scene Count. The vulnerable function (called “ProcessSFLData” for simplicity) first reads in the “Scene Count” value, performs some basic validation, and then allocates a structure to read in the following scene records one by one.

```
.text:30087A42   call   SWF_GetEncodedInteger ; Scene Count
.text:30087A47   mov    edi, [ebp+arg_0]
.text:30087A4A   mov    [esi+4], eax
.text:30087A4D   mov    ecx, [ebx+8]
.text:30087A50   sub    ecx, [ebx+4]
.text:30087A53   cmp    eax, ecx
.text:30087A55   jg     loc_30087BB4
.text:30087A5B   test   eax, eax
.text:30087A5D   jz     loc_30087B0E
.text:30087A63   mov    ecx, [edi+20h]
.text:30087A66   push   3
.text:30087A68   push   3
.text:30087A6A   push   0Ch          ; nCount
.text:30087A6C   push   eax          ; nSize
.text:30087A6D   call   mem_Calloc
```

Please note the following observations. First, the encoded integer that was read in is stored at [esi+4], and is immediately compared against ([ebx+8] – [ebx+4]). This value represents the amount of data left in the file ([ebx+4] is an offset into a buffer, and [ebx+8] is the total size of that buffer). The use of "jg" indicates that the comparison is signed, so a negative "Scene Count" value will pass this check, because it will always be less than some positive amount of data remaining in the buffer. Obviously, a negative value will also pass the subsequent check for 0. The negative value is then passed to the mem_Calloc() function, which treats it as an unsigned value. However, upon initial inspection of the mem_Calloc() function, the allocation will clearly fail, since integer overflow is checked when multiplying its nCount and nSize parameters:

```
.text:301B14D0 mem_Calloc     proc near           ; CODE XREF:
sub_3003359F+58 p
.text:301B14D0                                   ;
sub_3003362C+27 p ...
.text:301B14D0
.text:301B14D0 nSize        = dword ptr  8
.text:301B14D0 nCount        = dword ptr  0Ch
.text:301B14D0 arg_8        = dword ptr  10h
.text:301B14D0 arg_C        = dword ptr  14h
.text:301B14D0
.text:301B14D0   push   esi
.text:301B14D1   mov    esi, [esp+nSize]
.text:301B14D5   push   edi
.text:301B14D6   mov    edi, [esp+4+nCount]
.text:301B14DA   mov    eax, esi
.text:301B14DC   mul    edi
.text:301B14DE   test   edx, edx
.text:301B14E0   ja    short loc_301B14E7 ; overflow, return NULL
.text:301B14E2   cmp    eax, 0FFFFFFF0h
.text:301B14E5   jbe    short loc_301B14EE
.text:301B14E7
.text:301B14E7 loc_301B14E7:                      ; CODE XREF:
mem_Calloc+10 j
.text:301B14E7   pop    edi
.text:301B14E8   xor    eax, eax
.text:301B14EA   pop    esi
.text:301B14EB   retn   10h
```

So, this allocation is guaranteed to fail. The vulnerability in the code is that there is no check for an allocation failure. Below is the code that follows the call to mem_Calloc():

```
.text:30087A6D   call   mem_Calloc
.text:30087A72   push   eax
.text:30087A73   mov    ecx, esi
.text:30087A75   call   sub_3004A766
.text:30087A7A   and    [ebp+arg_0], 0
.text:30087A7E   cmp    dword ptr [esi+4], 0
.text:30087A82   jle    short loc_30087AFA
.text:30087A84   xor    ebx, ebx
```

The function called immediately after mem_Calloc() performs a variety of tasks, the most important of which is to populate [esi] with the pointer returned from mem_Calloc(). In this case, that means setting [esi] to NULL rather than point to a valid location in memory. A loop is subsequently entered for reading in the scene records. Note that this loop is never executed since [esi+4], the integer that was read in earlier, is less than 0. Therefore, the loop is entirely skipped and execution resumes at 0x30087AFA:

```
.text:30087AFA loc_30087AFA:                    ; CODE XREF:
ProcessSFLData+56 j
.text:30087AFA   mov    eax, [esi+4]
.text:30087AFD   mov    ecx, [esi]
.text:30087AFF   lea    eax, [eax+eax*2]
.text:30087B02   lea    eax, [ecx+eax*4]
.text:30087B05   mov    ecx, [ebp+arg_8]
.text:30087B08   sub    ecx, [eax-0Ch]
.text:30087B0B   mov    [eax-4], ecx
```

The code above does roughly the following:

```
struct _scene_record *record;

record = esi_struct->record_ptr + esi_struct->scene_count;

// or, without pointer arithmetic:
// char *pointer = (char *)esi_struct->record_ptr +
//      esi_struct->scene_count * 12;

record--;
record->offset_8 = (DWORD)arg_8 - record->offset_0;
```

So, if "pointer" is NULL, the address being written to can be chosen by the attacker by specifying an integer of the format: (0x80000000 | ((address + 4)/ 12)). Note that there is a restriction on which address an attacker may write to: it must be an address that is evenly divisible by 12 when 4 is added to it.

The value written to this location (denoted as arg_8 in the pseudocode above) is primarily the unsigned 16-bit "Frame Count" value read from the SWF file header, and so is controllable by the user. "Frame-Count" is a short but is written as a DWORD, and is also subtracted from by another value in memory, thus complicating exploitation.

Exploiting the Bug

Usually when exploiting an arbitrary memory overwrite vulnerability, the easiest and most direct way to seize execution control is to overwrite a function pointer of some kind. However, in this scenario, writing a valid pointer to a useful location in memory is difficult because the value being written is not exactly a value of the attacker's choosing. To further complicate exploitation, the memory location being written to is not completely arbitrary. It must be evenly divisible by 12 when 4 is added to it (which also implies it must be DWORD aligned – no partial pointer overwrites). Therefore, an alternate strategy will likely be more useful in exploiting this vulnerability. The attack chosen to exploit this bug utilizes the ActionScript Virtual Machine (AVM) in order to gain execution, and will be discussed once some of the basics have been covered.

Basics

Some basics about the AVM mechanics will help explain how exploitation is achieved. The details provided are specific to the ActionScript Virtual Machine 2 Virtual Machine (AVM2), which is what ActionScript 3.0 code is compiled for.

Flash Players that support ActionScript 3.0 and later (Flash Player versions 9 and up) implement a full virtual machine in which ActionScript code is verified and run. This contrasts with some of the earlier versions, where commands were compiled as a series of records that were parsed and executed sequentially. ActionScript "Files" are supplied as a blob of binary data encapsulated in a "DoABC" tag within an SWF file (tag type 0x52). This ActionScript file format – as well as other details about the virtual machine's functionality – is documented in some detail by Adobe, and is available at http://www.adobe.com/devnet/actionscript/articles/avm2overview.pdf. Quite a few structures need to be present for a typical ActionScript file, but such details do not pertain to the discussion of exploitation. Therefore, the focus will be primarily on the ActionScript bytecode itself, except in circumstances where further explanation is required.

ActionScript files contain methods, which in turn contain bytecode that is to be executed in an AVM2 environment. The bytecode undergoes two distinct phases: verification and interpretation. The verifier is responsible for ensuring that the code is legitimate and well-formed, and the interpreter simply parses each instruction and executes whatever operation the instruction implies. Each method has a variable number of "local registers," an evaluation stack for storing operand data as well as results, and a scope stack for holding scope objects. The amount of local registers and size of the two stacks are indicated in a method_body_info structure specific to the method in question. This structure, defined in the aforementioned Adobe AVM2 document, is as follows:

```
method_body_info
{
      u30 method
      u30 max_stack
      u30 local_count
      u30 init_scope_depth
      u30 max_scope_depth
      u30 code_length
      u8 code[code_length]
      u30 exception_count
      exception_info exception[exception_count]
      u30 trait_count
      traits_info trait[trait_count]
}
```

Various instructions in the AVM2 language allow access to specific elements of the register list and the two stacks. Interestingly, the interpreter function responsible for executing ActionScript bytecode allocates the local registers, the evaluation stack, and the scope stack contiguously on the runtime stack in the ActionScript3_Execute() function. The relevant code is shown:

```
.text:30184CF0   call   SWF_GetEncodedInteger_0
.text:30184CF5   mov    [ebp+6Ch+locationBase], eax
.text:30184CF8   lea    eax, [ebp+6Ch+var_40]
.text:30184CFB   push   eax
.text:30184CFC   call   SWF_GetEncodedInteger_0
.text:30184D01   mov    esi, eax
.text:30184D03   lea    eax, [ebp+6Ch+var_40]
.text:30184D06   push   eax
.text:30184D07   call   SWF_GetEncodedInteger_0
.text:30184D0C   mov    edi, eax
.text:30184D0E   lea    eax, [ebp+6Ch+var_40]
.text:30184D11   push   eax
.text:30184D12   call   SWF_GetEncodedInteger_0
.text:30184D17   sub    eax, edi
.text:30184D19   mov    edi, eax
.text:30184D1B   lea    eax, [ebp+6Ch+var_40]
.text:30184D1E   push   eax
.text:30184D1F   mov    [ebp+6Ch+var_A8], edi
.text:30184D22   call   SWF_GetEncodedInteger_0
.text:30184D27   mov    eax, [ebp+6Ch+var_40]
.text:30184D2A   mov    [ebp+6Ch+var_48], eax
.text:30184D2D   lea    eax, [edi+esi]
.text:30184D30   add    eax, [ebp+6Ch+locationBase]
.text:30184D33   add    esp, 14h
.text:30184D36   shl    eax, 2
.text:30184D39   add    eax, 3       ; (max_stack + local_count
+ (max_scope_depth - init_scope_depth)) * 4 + 3
.text:30184D3C   and    eax, 0FFFFFFFCh
.text:30184D3F   call   __alloca_probe
```

[Note: Astute readers will notice that there appears to be an integer overflow in the allocation of these structures. However, things aren't quite what they seem. The integers used for the size calculation of this allocation are validated prior to executing this function (to some extent, sort of…).]

After this function performs a few more initialization tasks, it executes the bytecode for the given method. Instructions that access local registers typically refer to the register by index, starting from 0. Instructions that access the stack(s) usually access elements on the top of the stack implicitly (or put new elements at the top of the stack). Either way, no verification is performed with regard to where data is being placed or read from by these instructions, because the verifier has already done that job. Specifically, it has guaranteed that local register indexes are in bounds, and also that there are no code paths where the stack can be inconsistent when a given instruction is executed (i.e. There are adequate elements on the stack for a given instruction. The values on the stack are of the correct types, and placing a new value on the stack won't overflow it). Needless to say, if there was a way to execute AS3 instructions that had never been verified, it would be quite dangerous. Unverified instructions would be able to manipulate the native runtime stack – made possible by specifying invalid local register indexes, or pushing/popping too many values from the AS3 evaluation stack.

Importantly, each value in a register or on a stack is stored as a DWORD which contains both typing information and a value. So, using instructions that write values to the "local registers" or the stack will result in an encoded data value being written to the destination location which includes typing information (the bottom three bits are reserved for this typing information). For example, consider the instruction:

```
pushshort 0x80
```

The resultant DWORD this instruction will write to the stack will be ((0xFFFFFF80 << 3) | 0x06).

Breaking the Verifier

Now that the AVM2 basics have been explained, an exploitation methodology can be developed. The attack works by manipulating a data structure used by the AVM2 verifier such that it doesn't correctly verify the ActionScript instructions for a given method. As mentioned in the previous section, the ability to execute unverified AS instructions has several interesting consequences, one of which is being able to read or write data values off of the Flash application's runtime stack. By overwriting EIP or some other useful pointer, control of execution flow is possible.

The ActionScript3 verifier performs a variety of tasks. It ensures that instructions and their parameters are well-formed and legal, performs dataflow analysis to verify the integrity of the evaluation stack, and completes additional tasks outside the scope of this document. The analysis of instructions and their operands are done in a loop, which looks roughly like this:

```
#define TOP_BITS_SET(x) ((x) & 0xC0000000)

unsigned char *codePtr, *codeEnd;
DWORD opCode, instruction_length = 0;

for(codePtr = buffer, codeEnd = buffer + length; codePtr < bufferEnd;
codePtr += instruction_length)
{
        DWORD arg_dword1, arg_dword2, arg_short, arg_byte;
        unsigned char *tmpPtr = codePtr;

        instruction_length = 0;
        opCode = *codePtr;

        if(AS3_ArgMask[opCode] == 0xFF)         // unused opcode
        {
              .. throw exception ..
        }

        .. check exception table stuff ..

        arg_dword1 = arg_dword2 = arg_short = arg_byte = 0;

        opcode_getArgs(&tmpPtr, &arg_dword1, &arg_short,
              &arg_dword2, &arg_byte);

        if(opCode == 0x25 || ( TOP_BITS_SET(arg_dword1|arg_dword2)
              && (opCode != 0xEE || TOP_BITS_SET(arg_dword2))))
        {
              .. throw exception ..
        }

        instruction_length = tmpPtr - codePtr;

        codePtr += instruction_length;

        if(codePtr > codeEnd)
        {
              .. throw exception ..
        }

        switch(opCode)
        {
              .. verify operands for individual instructions ..
        }
}
```

Many irrelevant details have been trimmed from the previous sample. Of particular interest is the opcode_getArgs() function, which looks something like this:

```
void opcode_getArgs(unsigned char **dataPtr, DWORD *arg_dword1, DWORD
*arg_short, DWORD *arg_dword2, DWORD *arg_byte)
{
        unsigned char *ptr = *dataPtr;
        DWORD opCode = *ptr++, mask = AS3_argmask[opCode];

        // only 2 operations with 1 byte operand
        if(opCode == INS_PUSHBYTE || opCode == INS_DEBUG)
        {
                *arg_byte = *ptr++;
                mask--;
        }

        // if mask is <= 0, all done
        if(mask <= 0)
        {
                *dataPtr = ptr;
                return;
        }

        // instructions with a 24-bit signed operand
        if(opCode >= INS_IFNLT && opCode <= INS_LOOKUPSWITCH)
        {
                char msb = ptr[2];       // will sign-extend
                *arg_short = (msb << 16) | *(unsigned short *)&ptr[0]
                ptr += 3;
        }
        else
                *arg_dword1 = SWF_GetEncodedInteger(&ptr);

        if(opCode == INS_DEBUG)
        {
                ptr++;
                mask--;
        }

        if(mask > 1)
                *arg_dword2 = SWF_GetEncodedInteger(&ptr);

        *dataPtr = ptr;

        return;
}
```

As seen, the length of an instruction depends on what operands a given opcode is deemed to have, which is calculated by checking the opcode itself, as well as a byte array named AS3_argmask. The AS3 interpreter does not work out instruction lengths the same way; it has a switch statement based on the opcode value and interprets each opcode specifically. If an unknown opcode is encountered, the interpreter silently skips the unknown instruction.

So, by overwriting a portion of this AS3_argmask array, it is possible to make an instruction appear longer or shorter than it really is to the verifier. The result is a desynchronization between what code the verifier sees and what code is actually executed by the AVM2 interpreter. There are two different ways that desynchronization could be achieved:

- *Changing an opcode mask from zero to non-zero: a variable number of trailing bytes will be interpreted as operands by the verifier, but as new instructions by the AVM2 interpreter.*
- *Changing an opcode mask from non-zero to zero: trailing bytes will not be interpreted as operands by the verifier when they should be, and the AVM2 interpreter will correctly use them as operands. This has a similar net effect: new instructions will begin at locations in the byte stream that the verifier has not correctly parsed and ensured to be safe.*

Using either of these methods will create the opportunity for executing unverified ActionScript code.

When selecting a target mask to overwrite, keep in mind that four masks are going to be corrupted, since a DWORD is written. Care should be taken to ensure that other legitimate AS3 methods will be disrupted as little as possible (Flash internally executes a significant number of AS3 methods, and if they don't function correctly, the malicious method supplied in this SWF file will never be reached.). With this in mind, the ideal targets in AS3_argmask array are unused bytecodes – those with an 0xFF mask. Recall from the disassembly shown in the last section that encountering unused opcodes during verification would usually result in an exception being thrown. However, by overwriting the mask value with anything that is not 0xFF, the verifier will deem it to be a valid opcode and read trailing bytes as parameters. As previously explained, the interpreter will silently skip a single byte, and thus all of the trailing bytes that the verifier deemed to be operands will be executed as instructions.

[Note: the exact amount of "trailing bytes" that are decoded as operands is variable, since integer operands can be anywhere between one and five bytes; therefore, the values that those bytes take on are significant.]

Constructing ActionScript ByteCode

Since overwriting values in the AS3_argmask array can result in executing unverified AS3 bytecode, a useful bytecode stream needs to be constructed. The AS3 bytecode should consist of a series of "marker" opcodes (that is, the unused bytecode that has had its mask value overwritten in AS3_argmask) followed by unverified instructions that are useful to execute. Ideally, the goal is to achieve the following steps:

1. Save the original EIP
2. Replace it with a pointer to data controllable by the user
3. Have ActionScript3_Execute() return
4. Have shellcode save register state and stack location
5. Allow shellcode to execute whatever it wants
6. Have shellcode restore original EIP on the stack and register context
7. Allow flash to return like nothing ever happened

As stated earlier, arbitrary DWORD values cannot be written to the "local registers" unmodified in AS3, due to value/type encoding. Therefore, it is not possible to set EIP to an arbitrary pointer of the attacker's choosing.[1] It is possible, however, to move a value already on the stack to somewhere else on the stack unmodified. Moving a value from the stack to a register or between registers does not require that the value be decoded and then re-encoded – that would be redundant. Moving values around on the stack is very interesting for two reasons:

1. It is possible to move the saved EIP on the runtime stack for ActionScript3_Execute() to elsewhere on the stack.

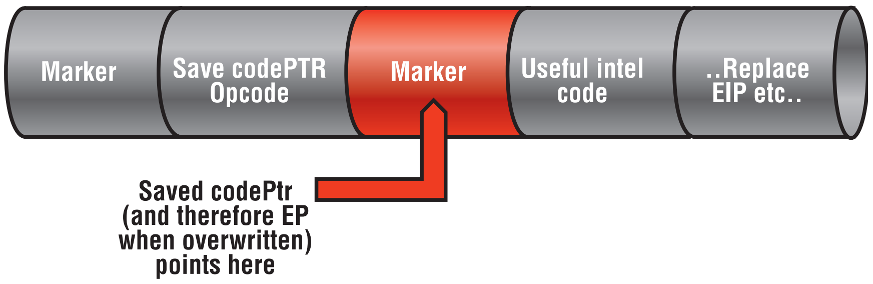
2. It is possible to move another pointer already on the stack over EIP, without corrupting anything else.

The ability to perform both of these operations will allow construction of a bytestream that achieves the steps mentioned earlier. For the second part, a pointer must exist on the stack that can be moved over the saved EIP of ActionScript3_Execute(). As it happens, there is such a pointer on the stack, pointing to data the user can control – namely a "codePtr" variable, which is pointing to the AS3 bytecode being currently executed.

[1] *It is possible to write an integer with the top three bits cleared, and when it is left-shifted it forms a valid address. But this would add the reliance on some fixed address where shellcode resides, which should be avoided if possible. Also, some data types are actually encoded as pointers, but using them would be a little complicated. The biggest problem is that these pointers usually point to data structures that contain pointers to user controlled data. Additionally, because the lower three bits include typing information, the resultant pointer would point several bytes into this data structure.*

The Dual Bytestream Problem

In order for the attack to succeed, the data pointed to by the "codePtr" variable must point to a bytestream that contains valid intel instructions as well as valid AS3 bytecode. Although the AS3 interpreter doesn't validate the bytestream at all, recall that the verifier has validated it to some extent (except for the "trailing bytes" of each marker byte that has been inserted). The code also must not crash the interpreter (register indexes that are too far out of bounds would cause such a crash). When EIP is replaced with codePtr, the bytestream will look something like this:



Since codePtr points to the marker byte that was chosen, the marker must be a valid intel opcode that will not cause the application to crash at the very least, and not affect register state much either. Add to this the address restrictions mentioned previously (address + 4 must be divisible by 12), and the range of options appears limited. One suitable candidate for Flash9d/IE is using the marker byte 0xF5, which is the intel CMC instruction. It has the following properties:

1. The address that needs to be overwritten is 0x302047C4, which is divisible by 12 when 4 is added to it.

2. All four AS3 bytecodes 0xF4, 0xF5, 0xF6, and 0xF7 are unused, so overwriting those four masks should not affect legitimate AS3 bytecode.

3. The intel CMC instruction is benign – all it does is flip the CF bit in the EFLAGS register.

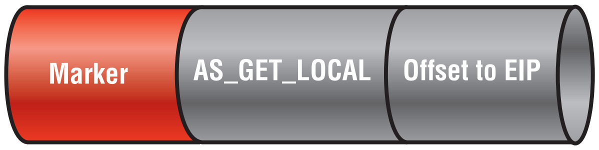
Perfect. For Flash9e/IE, the plugin is positioned slightly differently in memory, such that the alignment for AS3_argmask is different and 0xF5 cannot be used. However, the 0xF8 or 0xF9 marker may be used (the intel instructions CLC and STC, respectively). As for Firefox, the 0xFC marker value is usable (the intel CLD instruction) for Flash9d, and the 0xF5 marker (CMC) for Flash9e.

The Final ByteStream

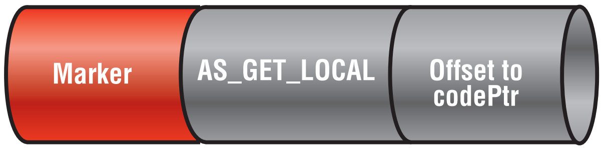
A marker byte has been selected, and the goals have been set, so the bytestream needed should be growing clearer. Before an example bytestream is shown, a few AS3 instructions that will be utilized are illustrated.

Bytecode Name	**Explanation**
Marker	The marker byte that has been chosen. The following few bytes will be incorrectly marked as operands by the verifier but will be executed as instructions by the interpreter.
AS_GET_LOCAL	Get a value from a local register. This opcode takes one operand – a local register index (which is used in a memory calculation as a source for data: register_memory_pointer + index * 4). Since this will be executed in an unverified context, it requires getting any arbitrary value off the stack. The value read from the "local register" will be placed on the AS3 evaluation stack.
AS_SET_LOCAL	The compliment of AS_GET_LOCAL. This writes a value to a "local register," which again will be an out of bounds location in this scenario.
AS_POP	Pop a value from the AS3 evaluation stack.
AS_NOP	Do nothing.
AS_RETURN_VOID	Return from the current method, which will cause ActionScript3_Execute() to finish its work and return to the caller.

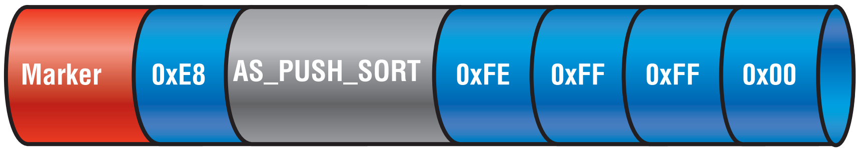
The bytestream is now presented in several chunks, with accompanying explanations.



Use an unverified AS_GET_LOCAL command to get the saved EIP and place it on the AS3 evaluation stack.



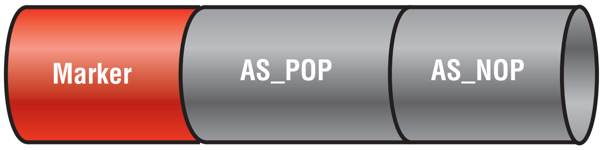
Use an unverified AS_GET_LOCAL to get the codePtr value off the stack and place it on the AS3 evaluation stack.



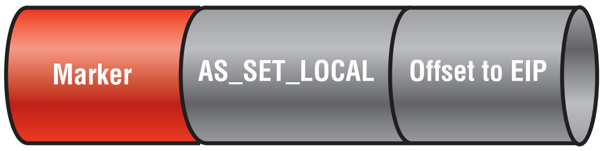
This one requires some explanation.

From the AS3 interpreter's point of view, the marker byte will be treated as a NOP, and so will 0xE8, because it is an undefined instruction. Then the AS_PUSH_SHORT instruction will be encountered, and the remaining bytes will be interpreted as an encoded integer which will be pushed on to the AS3 evaluation stack.

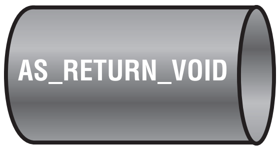
From the native intel point of view, this is the location that will be pointed to when EIP is replaced. The marker byte will do nothing effectively, and the following 0xE8 byte will cause a jump to elsewhere in memory. In this case, backwards, to where some shellcode exists. More explanation on why shellcode exists there will be explained in the "Insert Shellcode" section later.



This just pops the bogus short that was pushed on to the AS3 evaluation stack in the previous step. The evaluation stack now has the saved codePtr value at the top.



This overwrites the saved EIP with the codePtr value which is at the top of the AS3 evaluation stack. Notice that the evaluation stack is not empty; it still contains the original EIP. Therefore, there is a saved copy of the original EIP on the stack when the shellcode executes, and its relative location on the stack is known. Nothing else has been corrupted.



Finished! Let ActionScript3_Execute() return, and control of the application will be seized.

Putting It All Together

All of the necessary information for producing an exploit using this methodology is now known. The procedure for performing the attack is as follows:

Step 1. Perform Overwrite

Triggering the bug is the easy part. It simply involves constructing a "DefineSceneAndFrameLabelData" tag with a negative scene count which, when multiplied by 12, will point to the relevant address in the AS3_argmask table. Specifically, the formula for writing the correct "Scene Count" value is:

```
address = AS3_argmask_address + (marker_byte - (marker_byte % 4)) + 4
scene_count = (0x80000000 | (address / 12))
```

Recall that the value written to this address will be based on the 16-bit "Frame Count" value in the SWF file header, which can be pretty much anything, as long as the mask being overwritten ends up being a positive value.

[Note: As mentioned previously, the fact that a DWORD is overwritten means that four masks in the AS3_argmask table are going to be overwritten, so some care must be taken to ensure that all four masks being corrupted will not affect properly formed AS3 methods. By having a reasonable size frame_count value in the header, the two masks that make up the least significant bytes of the DWORD can be easily made positive values. For the other two masks, it is dependant on what "DWORD" is at the location eight bytes before the memory location being written to. In the end, this is not a significant hurdle.]

Step 2. Insert Bogus DoABC Tag

Generating a well-formed "ActionScript" file is required for this step, which contains an AS3 method with a bytestream similar to the one discussed earlier. Again, the specifics of this format are recounted at the following link: http://www.adobe.com/devnet/actionscript/articles/avm2overview.pdf.

Step 3. Insert Shellcode

Recall that when discussing the malicious bytestream, a backwards branch was performed to some shellcode which happened to be before the AS3 bytecode in memory. The explanation for this is simple: the AS3 bytecode is not copied out of the original SWF file buffer when it is processed. The pointer to the code is pointing to a location within the malicious file (which has been entirely read into memory). Therefore, to have shellcode just before the AS3 bytecode in memory, an SWF tag must precede the DoABC tag that is filled with shellcode. A wide variety of tags could be used for this, one example being the DefineBits tag (tag 0x06).

So, the completed SWF file would look something like this:

[SWF Header]
[Corrupt DefineSceneAndFrameLabelData tag]
[DefineBits tag with shellcode]
[DoABC tag with malicious bytecode]
[ShowFrame tag]

Notice the last tag in this file is a "ShowFrame" tag (tag 0x01). This is required for the methods in the DoABC tag to be executed.

Reliability Notes

The methodology presented is quite robust. It relies on a single address – the address of the AS3_argmask table. The EIP redirection, address of shellcode, and fixup code for restoring EIP require no hard-coded addresses of any kind. Additionally, the method of utilizing the AVM to gain execution has a low impact on the rest of the process. Apart from unused AS3_argmask values being corrupted, gaining execution from that point does not damage any other structures or variables. EIP can be neatly overwritten without affecting anything else in the function and maintaining register state. This makes it possible to fully recover the process state after the shellcode has executed without risking the browser crashing.

There are several factors that also affect exploitation reliability that have not been touched on thus far. They are addressed below.

Internet Explorer/Firefox Parallel Exploitation

Microsoft® Internet Explorer (IE) and Firefox use different binaries for flash, due to their differing plugin architectures. The binaries contain identical code except for the glue code to communicate with the browser. Interestingly, the base address of the IE version and Firefox version are identical. Some testing was done for generating SWF files that exploited the vulnerability multiple times (by containing multiple corrupt "DefineSceneAndFrameLabelData" tags). Experiments proved that the data section for IE and Firefox were closely aligned such that it was possible to overwrite both the Firefox and IE target addresses in a single file, without causing a crash in either browser. Put another way, it is possible to generate an SWF file that successfully exploits both IE and Firefox for a given flash version without crashing either one. This was verified with both browsers for the Flash9d and also the Flash9e plugin. Previous versions were not tested but are expected to exhibit the same behaviour.

Vista and ASLR

Vista's ASLR features require that the binary is compiled with the /dynamicbase switch available on recent Microsoft compilers. Essentially, using this switch sets a flag in the PE header (0x40 in the DllCharactersitics member of the optional header) that will indicate that the binary should receive a random base address when loaded. Since flash does not use this switch, ASLR does not cause the Flash DLL to be moved in memory in Windows® Vista, and hence can still be reliably exploited. Combining this with the previous point, it is possible to generate an SWF file that will reliably exploit both IE and Firefox on all recent versions of the Windows operating system, including Vista.

Conclusion

Memory corruption vulnerabilities are now being exploited using application-specific attacks, like the scenario leveraging the ActionScript Virtual Machine. Learning how this attack works will help refine countermeasures to protect this and other similar vulnerabilities. For more information about the flash vulnerability, please visit http://www.iss.net/threats/289.html to read the complete security advisory prepared by the IBM Internet Security Systems research and development team.