Interpreter Exploitation: Pointer Inference and JIT Spraying

Dion Blazakis <dion@semantiscope.com>

Abstract

As remote exploits have dwindled and perimeter defenses have become the standard, remote client-side attacks are the next best choice for an attacker. Modern Windows operating systems have quelled the explosion of client-side vulnerabilities using mitigation techniques such as data execution prevention (DEP) and address space layout randomization (ASLR). This work will illustrate two novel techniques to bypass DEP and ASLR mitigations. These techniques leverage the attack surface exposed by the advanced script interpreters or virtual machines commonly accessible within the browser. The first technique, pointer inference, is used to find the memory address of a string of shellcode within the ActionScript interpreter despite ASLR. The second technique, JIT spraying, is used to write shellcode to executable memory by leveraging predictable behaviors of the ActionScript JIT compiler bypassing DEP. Future research directions and countermeasures for interpreter implementers are discussed.

Introduction

The difficulty in finding and exploiting a remote vulnerability has motivated attackers to devote their resources to finding and exploiting client side vulnerabilities. This influx of different client side attackers has pushed Microsoft to implement robust mitigation techniques to make exploiting these vulnerabilities much harder. Sotirov and Dowd [1] have described in detail each of the mitigation techniques and their default configurations on versions of Windows through Windows 7 RC. Their work shows some of the techniques available to bypass these protections and how the design choices made by Microsoft has influenced the details of these bypasses. One thing that stands out throughout this paper is how ripe a target the browser is for exploitation – the attacker can use multiple plug-ins, picking and choosing specific exploitable features, to set-up a reliable exploit scenario.

The classic web browser, bursting at the seams with plug-ins, could not have been designed with more exploitation potential. It requires a robust parser to parse and attempt to salvage 6 versions of mark-up. With the advent of "Web 2.0", a browser must now include a high performance scripting environment with the ability to rewrite those parsed pages dynamically. The library exposed to the scripting runtime continues to grow. Additionally, most browsers are now taking advantage of recent JIT and garbage collection techniques to speed up Javascript execution. All this attack surface and we haven't begun to discuss the plug-ins commonly installed.

Rich internet applications (RIAs) are not going away and Adobe currently maintains a hold over the market with Flash – the Flash Player is on 99% of hosts with web browsers installed. Sun's Java Runtime Environment is another interpreter commonly installed. Microsoft Silverlight is an RIA framework based upon the .NET runtime and tools. Silverlight is still struggling to gain market share but could be a contender in the future (e.g. Netflix On-Demand is starting to use this technology). Each of these plug-ins require a complex parser and expose more attack surface through a surplus of attacker reachable features. For example, Adobe Flash Player implements features including a large GUI library, a JIT-ing 3D shader language, a RMI system, an ECMAScript based JIT-ing virtual machine, embeddable PDF support, and multiple audio and video embedding or streaming options. All of this is available by default to the lucky attacker.

Considering this, it is worth putting in time and effort to develop application specific techniques that will help exploit vulnerabilities within this browser ecosystem. DEP and ASLR are very real thorns in the side of an exploit developer. DEP makes locating shellcode difficult; the attacker must find a page with executable permission **and** find a way to write to it when it has writable permission **and** figure out the location. Alternatively, the attacker could find some code to reuse – as in the return-oriented programming based attacks [3,4]. ASLR throws a wrench into this plan by obfuscating the base address of the loaded images. See Sotirov and Dowd [1] for a detailed explanation on the implementation of ASLR and DEP in various Windows operating systems. For now, we'll assume ASLR is close to perfect – the attacker cannot guess the location of a loaded image, the heap, or the stack.

This paper focuses on two general techniques to thwart these mitigations: pointer inference and JIT spraying. Pointer inference is the act of recovering the memory address of an internal object via "normal" (i.e. not through an exploit) interactions with the target software. Adobe Flash Player is used as an example. I will present and walk through a script to determine the address of a string object within Flash Player's ActionScript interpreter. JIT spraying is similar to heap spraying; the attacker will allocate many executable pages of attacker influenced data. I show how to construct an ActionScript script that, when JIT compiled by the ActionScript engine, lays out stage-0 shellcode to load and execute the next stage of shellcode (located in an ActionScript string). Next, I discuss the possibility of using the two techniques together to attack a Windows Vista target. Finally, some countermeasures and future research directions are discussed.

Pointer Inference

Getting from control of EIP to exploit is much more difficult when the address space is randomized. Even getting from a proof-of-concept crash to control of EIP may be difficult. Proving a vulnerability is sometimes easily accomplished using a heap spray to place an attacker constructed structure at a known address. Unfortunately for the exploit developer, heap sprays are not always reliable (and are certainly not always possible – if the attacker cannot allocate large heap objects). Additionally, heap spray mitigation techniques have already appeared in the academic world [10] and Microsoft has a simple mitigation (mapping the commonly used heap spray pages at process start-up) in their EMET tool [9]. Reliable exploits are within range with the ability to derive the heap address of a runtime object dynamically.

Scripting environments are a perfect target for pointer inference – the objects usually live on the heap and the runtime language and libraries provide multiple ways to manipulate and inspect objects. Additionally, scripting languages tend to be dynamically typed, so the built in container objects are often heterogeneous – objects stored by reference are often stored in the same structure as those stored by value. The goal is to find a way to determine the memory address of a script object in the interpreter/virtual machine. Extracting the location of a Javascript string containing shellcode, while not the only use of the technique, would be a good exploit building block. Target VMs include (ECMA|Java|Action)Script, Java, Python, Ruby, PHP, and the .NET CLR. The Javascript engine in the browser, the Javascript engine in Adobe Reader, and the ActionScript engine in the Adobe Flash Player are all available from within most browser installations (if they have the Adobe plug-ins installed and don't have Javascript disabled).

For this paper, I will show how to derive the memory address of an object in the ActionScript virtual machine using the ordering of objects when iterating over an ActionScript Dictionary object.   I've chosen to show this proof of concept using the Flash Player for multiple reasons: I've spent more time reversing it than the others, it is cross platform – regardless of browser it is the same code base for the plug-in, and, last but not least, Adobe has released the source to a fork of their engine [11]. To understand the details of the technique, I will first describe how the objects are stored internally by the interpreter and then how the built-in Dictionary container is implemented by the interpreter.

Internal Representation of ActionScript Objects

Before discussing ways of disclosing the address of an ActionScript objects, let's discuss how the objects are stored internally by the interpreter. When an ActionScript object is created by the executing script, the interpreter acts based upon the type of the object. If the object is a small primitive object, such as an integer or boolean, it is stored by value. Objects such as doubles, strings, or class instances will be stored by reference – the interpreter will allocate a buffer large enough to hold the object and store a pointer to this value.

ActionScript is a dynamically typed language. Dynamically typed languages do not assign types to values at compile time. Instead of building type annotations into the language and enforcing typing constraints at compile time, these languages provide runtime functions for examining and comparing object types. The interpreter then ensures that all operations are valid for the operands used at the time of the operation. When the interpreter detects a type mismatch (e.g. an instance is queried for a method it does not implement), ActionScript will either throw an exception or perform an implicit coercion. This runtime typing requires the object heap to store both the type information and the value.

To handle this runtime typing requirement, the ActionScript interpreter represents internal objects using tagged pointers – internal, this object is called an "atom". Tagged pointers are a common implementation technique to differentiate between those objects stored by value and those stored by reference using the same word sized memory cell. A tagged pointer stores type information in the least significant bits and stores a type specific values in the most significant bits. As shown in Illustration 1, the ActionScript atom is 32 bits wide; it allocates 3 bits to store the type information and uses 29 bits for the value.

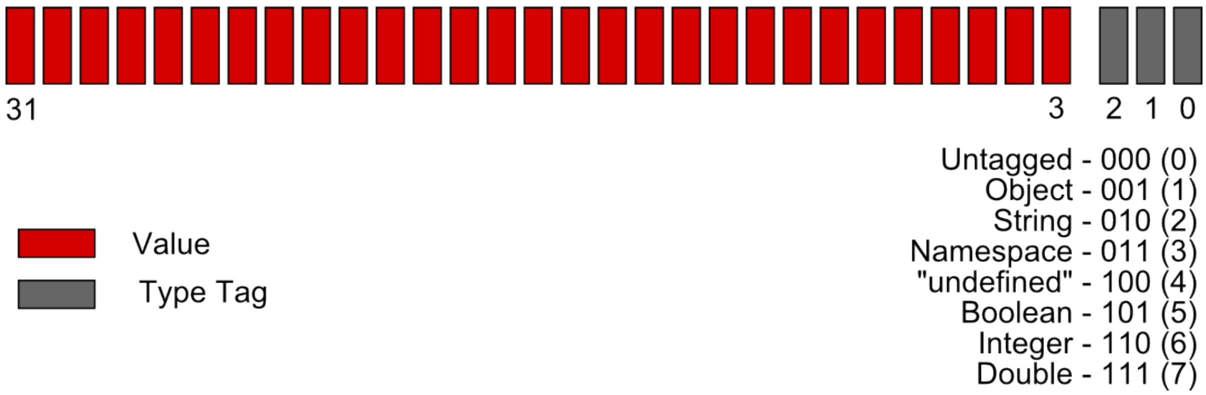


Illustration 1: Layout of an ActionScript atom

Some examples might help illustrate how this works. Take the following ActionScript declarations:

```
var x = 42; // Integer atom
var y = “blackhat”; // String atom
var z = new Dictionary(); // Object atom
var b = true; // Boolean atom
```

The variable `x` is a local variable holding the value `42`. The interpreter will create a mapping between the local variable within the current scope and the value `42`. As described above, the value will be stored internally as an atom. An integer atom, which can hold values between -2^{28} and $2^{28} – 1$, is created by shifting the value left 3 bits to make room for the type tag. The integer atom is tagged with a 6 as shown in Illustration 1. This process is shown in the Python session below:

```
>>> def atomFromInteger(n):
  return (n << 3) | 6

>>> '0x%08x' % (atomFromInteger(42),)
'0x00000156'
```

The String and Object atoms are “reference” atoms – they store pointers to garbage collected memory on the interpreter heap. Converting the `y` and `z` variables to atoms requires first allocating a block of memory to store the value and then creating the atoms using the memory address of the actual value. Below is an example of doing this for `z` in Python – the extra calls are mocked up, don't let them distract you.

```
>>> def atomFromObject(obj):
  return (obj & ~7) | 1

>>> a = actionScriptHeapAlloc(size_of_Dictionary)
>>> Dictionary.initialize(a)
>>> '0x%08x' % (atomFromObject(a), )
'0x00c8b301'
```

The goal of illustrating this internal representation is to be able to explain that both values and references (memory addresses / pointers) are used as atoms by the interpreter. Next, I will explain the use and implementation of the ActionScript Dictionary class.

Implementation of ActionScript Dictionaries

The built-in ActionScript Dictionary class exposes an associative map data structure. When used from within an ActionScript script, it provides an interface to associate any ActionScript object with any other ActionScript object as a key/value relation. The Dictionary object can then be queried using square brackets, similar to a Python dict. Additionally, the user can iterate over the dictionary to operate on each key/value pair. The order of the iteration is not specified by the definition of the API and is an implementation detail not to be relied upon. Example use:

```
var dict = new Dictionary();

dict["Alpha"] = 0x41414141;
dict[true] = 1.5;

var k;
for (k in dict)
{
  f(dict[k]);
}
```

Internally, Flash Player's Dictionary class is implemented using a hashtable. The hashtable derives the hash from the key atom and stores the key and value atom together in the table. When iterating over the Dictionary, the hashtable is walked from lowest to highest hash value. The last details are how the hash function works, how collisions are resolved, and how the hash table grows. The hash table is always a power of two in size – this is maintained for two reasons: the hash function now becomes a fast masking operation and the constants used by the quadratic probe rely on a power of two sized table. The hash tables grows to the next power of two when the number of empty cells in the table drops below 20% of the total size. To grow the hashtable, a new table is allocated and all entries are rehashed and inserted into the new table. This hashtable implementation discloses an ordering between integer (value) atoms and object (reference) atoms – the object atoms are compared directly to the integer atoms. The hash function will remove some of the most significant bits of the atoms but a large hashtable will use most of the bits. This ordering is used to disclose memory addresses of reference atoms (Objects, Strings).

Integer Sieve

Since integers are placed into the hashtable using their value as the key (of course, the any top bits will be masked off), we can determine the atom value of some ActionScript object by measuring where the new object is found when iterating over the hashtable. By recording the integers that fall before and after the newly inserted object, we can derive a bound on the atom of the new object. Since Object atoms are just pointers (with the first 3 bits modified), we can disclose as many bits of a pointer as we can grow the hashtable.

To avoid the problem of a hash collision, we perform the test twice: once with all even integers and once with all odd integers (up to some power of two – the larger, the more bits we discover). After creating the Dictionaries, we insert the victim object into both Dictionaries. The values associated with the keys stored in the Dictionary do not impact any of this – only the keys and their ordering are used. Next, search each Dictionary using the for-in construct, recording the last key visited and breaking when the current key is the victim object. We now have two integer values (the last integers before the victim object for both the even and the odd Dictionaries).

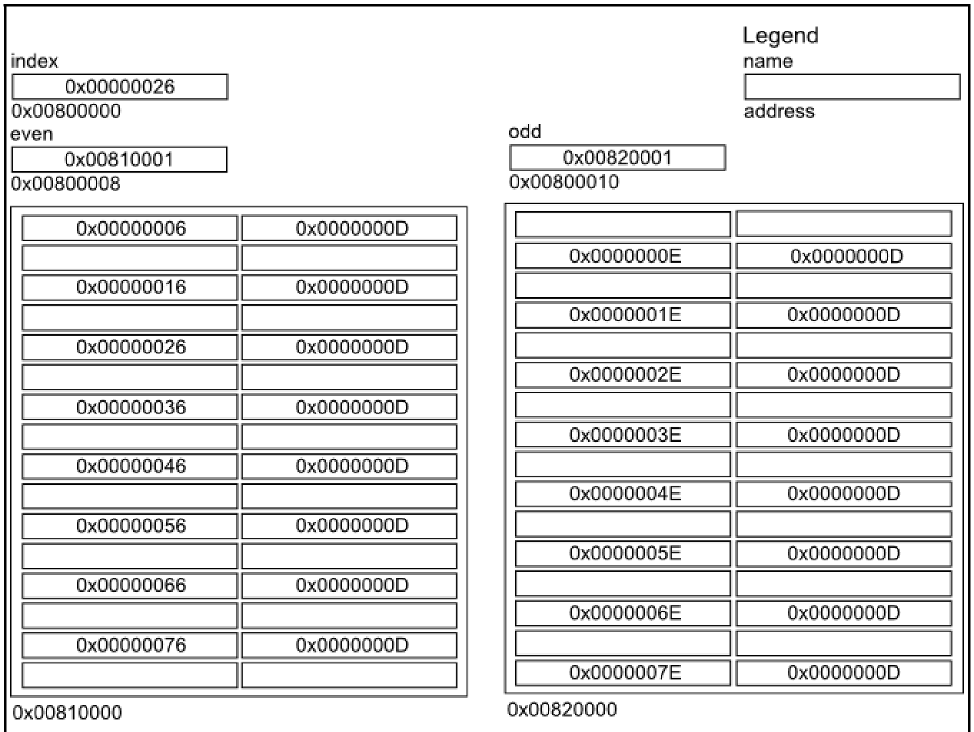
The two integers should differ by 17. This is due to the linear probe; when a hashtable insert collides, it uses a quadratic probe to find an empty slot. The first try is at (hash(victim_atom) + 8) which always collides – 2n + 8 = 2(n + 4) or (2n + 1) + 8 = 2(n + 4) + 1. The next try is (hash(victim_atom) + 17) which always succeeds – 2n + 17 = 2(n + 8) + 1 or (2n + 1) + 17 = 2(n + 9). The only way for the two integers to differ by anything other than 17 is if the probe wrapped. Otherwise, the smaller integer is the one from the Dictionary that didn't have the collision. When the difference isn't 17 (wrapped around), the larger value is from the Dictionary that didn't have the collision. We now have the integer that, when turned into an atom is 8 smaller than the victim atom. Finally, to get the victim atom from the integer, x: (x << 3 + 8) or more to the point ((x + 1) << 3).

That is a handful; let's walk through the execution:

```
// First, create the Dictionaries
var even = new Dictionary();
var odd = new Dictionary();

// Now, fill the Dictionary objects with the integer atoms
var index;
for (index = 0; index < 8; index += 1) {
  even[index * 2] = true;
  odd[index * 2 + 1] = true;
}
```

Here is what the heap will look like after inserting the integer atoms:



0x00000006	0x0000000D
0x00000016	0x0000000D
0x00000026	0x0000000D
0x00000036	0x0000000D
0x00000046	0x0000000D
0x00000056	0x0000000D
0x00000066	0x0000000D
0x00000076	0x0000000D

0x0000000E	0x0000000D
0x0000001E	0x0000000D
0x0000002E	0x0000000D
0x0000003E	0x0000000D
0x0000004E	0x0000000D
0x0000005E	0x0000000D
0x0000006E	0x0000000D
0x0000007E	0x0000000D

Illustration 2: Heap after Integer atoms are inserted

Next, insert the reference object to leak the address of:

```
var victim = “AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA”;

even[victim] = true;
odd[victim] = true;
```

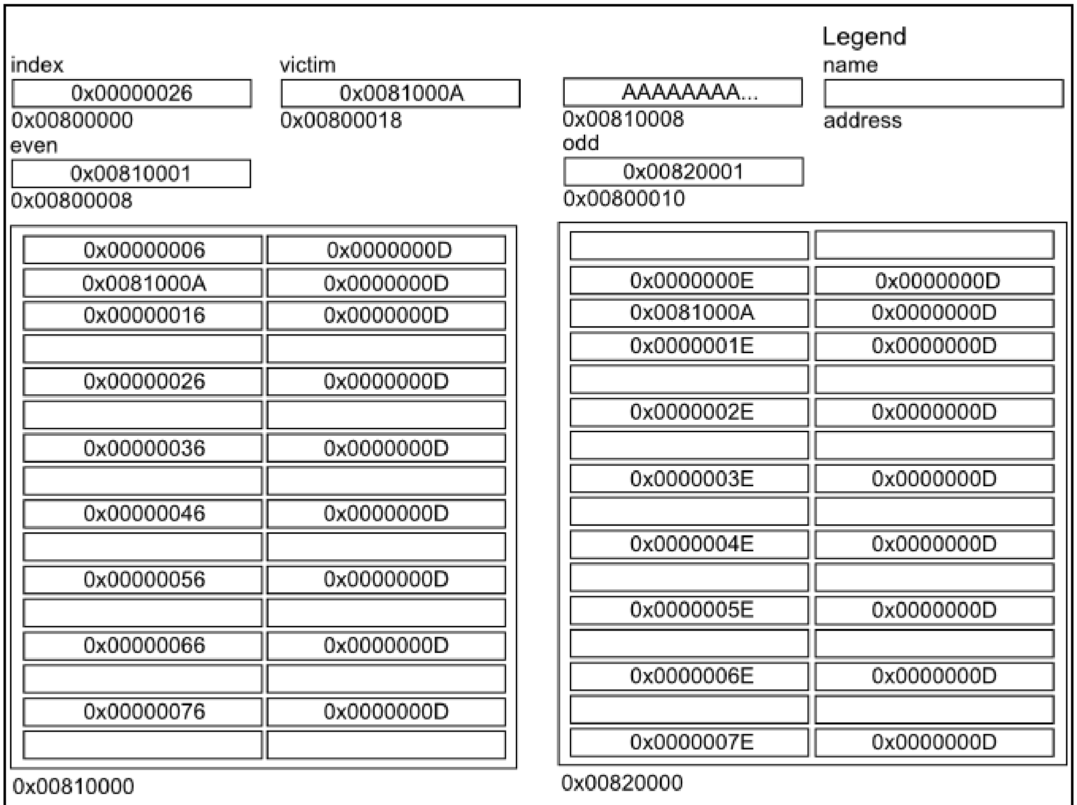


Illustration 3: Heap after adding the victim object to the sieves

In Illustration 3, you can see how the insertion of the victim object wraps around once (the table is 16 entries long, so on a collision using these integer sieves, 17 will be added to the index modulo 16 – just adding 1 to the index where the collision took place). The last step is to iterate over both Dictionaries recording the integer just prior to finding the victim object:

```
var curr, evenPrev, oddPrev;

for (curr in even)
{
  if (curr == victim) { break; }
  evenPrev = curr;
}

for (curr in odd)
{
  if (curr == obj) { break; }
  oddPrev = curr;
}
```

After executing this snippet, evenPrev will contain the value 0 (atom: 0x00000006) and oddPrev will contain the value 1 (atom: 0x0000000E). Let's pretend that evenPrev and oddPrev differ by 17 (and the tables were a more reasonable size – say, 4 million). evenPrev is smaller – this means we can derive the lowest bits of the victim address by adding 1 to evenPrev (the value, 0, which is what is exposed to us in ActionScript) and shifting left 3. This results in 0x00000008. That matches the lowest bits of the victim address.

Extracting memory address leaks is not unique and other vectors for leakage are not hard to dream up. If we restrict ourselves to data structure leaks, internal ids (see Python) and hash functions (as opposed to the ActionScript technique, where the hash is not available directly to the script; see .NET and Java). These functions need a unique value per heap object and the address is one such value that could be used.

I have included the source to a sample ActionScript package that performs this process and is verified to work for Adobe Flash Player plug-in (NPSWF32.DLL and Flash10c.ocx) versions 10.0.32.18 and 10.0.42.34.

JIT Spray

Data Execution Prevention (DEP) makes executing delivered shellcode quite difficult – the stack and default heaps are marked non-executable. As far as I know, the best public method to bypass it is to find a loaded DLL that is not ASLR enabled and use the available code to manufacture return-oriented shellcode [3,4] that turns off DEP for the process or allocates executable memory and copies the next stage of shellcode there.

Most modern interpreters implement a Just-In-Time (JIT) compiler to transform the parsed input or bytecode into machine code for faster execution. JIT spraying is the process of coercing the JIT engine to write many executable pages with embedded shellcode. This shellcode will entered through the middle of a normal JIT instruction. For example, a Javascript statement such as “var x = 0x41414141 + 0x42424242;” might be compiled to contain two 4 byte constants in the executable image (for example, “mov eax, 0x41414141; mov ecx, 0x42424242; add eax, ecx”). By starting execution in the middle of these constants, a completely different instruction stream is revealed.

The rest of this section explains one implementation of this for the Adobe Flash Player ActionScript bytecode JIT engine. The end result is a Python script and an ActionScript script. The Python script generates the ActionScript. The ActionScript, when loaded and after the bytecode engine performs a JIT compile, lays out an executable page on the heap. When this executable page is entered at a known offset (0x6A, for the given example code below) will execute stage-0 shellcode that marks the rest of the page RWX and copies the next stage of shellcode from an ActionScript string that can be modified before compilation.

The key insight is that the JIT is predictable and must copy some constants to the executable page. Given a uniform statement (such as a long sum or any repeating pattern), those constants can encode small instructions and then control flow to the next constant's location.

Development

By placing a breakpoint on the VirtualProtect calls in the Flash Player, we can witness the JITed code that is generated from the ABC bytecode (which is compiled ActionScript). By experimentation, I was able to determine that a long XOR expression (a ^ b ^ c ^ d ^ …) would be compiled down to a very compact set of XOR instructions.

For example, after JIT-ing:

```
    var y = (
      0x3c54d0d9 ^
      0x3c909058 ^
      0x3c59f46a ^
      0x3c90c801 ^
```

```
       0x3c9030d9 ^
       0x3c53535b ^
       ...
```

is turned into:

```
03470069    B8 D9D0543C     MOV EAX,3C54D0D9
0347006E    35 5890903C     XOR EAX,3C909058
03470073    35 6AF4593C     XOR EAX,3C59F46A
03470078    35 01C8903C     XOR EAX,3C90C801
0347007D    35 D930903C     XOR EAX,3C9030D9
03470082    35 5B53533C     XOR EAX,3C53535B
```

Now, note that if execution begins at 0x0347006A:

```
0347006A    D9D0            FNOP
0347006C    54              PUSH ESP
0347006D    3C 35           CMP AL,35
0347006F    58              POP EAX
03470070    90              NOP
03470071    90              NOP
03470072    3C 35           CMP AL,35
03470074    6A F4           PUSH -0C
03470076    59              POP ECX
03470077    3C 35           CMP AL,35
03470079    01C8            ADD EAX,ECX
0347007B    90              NOP
0347007C    3C 35           CMP AL,35
0347007E    D930            FSTENV DS:[EAX]
```

This is a popular GetPC method – using the floating point state save [8].

ActionScript allows the dynamic loading (and runtime generation) of bytecode – using this, we can force the repeated loading of a given bytecode file spraying the constructed JIT code on the heap. This is the JIT spray; with a reasonable guess and luck, the attacker can execute shellcode despite ASLR and DEP. In the last section, we will take a look at using the

In the sample given above, notice that the control from one DWORD to the next only takes up a single byte. By taking advantage of the single byte XOR EAX opcode, the CMP AL is a semantic NOP and doesn't require the two bytes a JMP 8bit-offset would take.

Putting It Together

One thing that was glossed over in the pointer inference discussion was the value that is pointed to by the leaked pointer. What does the address leak gain an attacker? What attacker controllable values are available via the leaked pointer? For Object atoms, the pointer points to a C++ instance that doesn't have any easily controllable fields. For String atoms, the pointer points to a C++ instance containing a length field and a pointer to the string buffer. This will certainly be useful in some instances, but it isn't a direct pointer to the string buffer. In the end, it's really just a pointer into the heap. To try and find a use for this pointer, we must understand the Flash heap and some of the details of Windows memory architecture.

Flash objects are allocated using a custom allocator which boils down to VirtualAlloc. When expanding the heap, the allocator tries to allocate the next chunk contiguously in 16MB increments. If that first allocation fails, it tries again without the start address hint – i.e. getting rid of the contiguous constraint. The pages for the JIT engine are allocated directly using VirtualAlloc by estimating the space needed by counting opcodes when loading the bytecode via an ABC file. The allocation for all methods/functions found in a bytecode file occurs up front (and will allocate this space on-demand when the first methods is called and compiled). VirtualAlloc will map pages at a 64KB granularity and does so with a linear scan finding the first hole that matches the size requested. With these details, we can come up with a use for the heap address primitive.

While dreaming up these techniques, the plan was to put the two together for a reliable bypass of DEP and ASLR. To do this, we must use the heap address primitive to determine the address of a JIT block. Here is the current method I've developed in the browser environment:

1. Open a SWF file that contains enough bytecode to force an allocation of 0x01010000 bytes.
2. Open a SWF file that sprays the heap with many small ActionScript objects.
3. Open a second SWF forcing an allocation of 0x00FF0000 bytes.
4. Remove the first large SWF file – this will deallocate/unmap the 0x1010000 bytes.
5. Spray more than 16MB of small ActionScript objects, keeping them in a linked list structure.
6. Load the SWF with the JIT Spray.
7. Iterate over the linked list structure at large intervals (0x00100000) recording the address of the object.
8. When an unsmooth edge is found in the recorded addresses, this marks the start of the newly mapped space. This new space should be the start of the memory used for the first SWF.
9. A tight bound can be obtained for the start address by refining the above search using successively smaller intervals and updating the bounds.
10. Finally, we know the JIT spray should have reached this address we've found in the previous step + 0x01000000.

This whole process is quite roundabout, but the Adobe Flash player doesn't seem to ever unmap memory. By not unmapping memory used by the object heap, the JIT engine will never allocate a page that was once used by the Object heap.

Once the address of a JIT sprayed block is found, the target EIP can be calculated and the exploit can be triggered. This process is quite reliable, but with the current speed of the address leak it takes between 5 and 10 minutes to trigger. If a different address leak were found this would be quicker. While testing this on a Windows Vista target with IE8, a straightforward JIT spray can be used and a predictable address is not hard to find. In a browser with a lot of memory activity, the longer method would probably be more reliable, but more testing is needed.

Bibliography

Protection Bypasses

[1] Alex Sotirov and Mark Dowd. Bypassing browser memory protections in Windows Vista. BlackHat USA 2008.

[2] Tyler Durden. Bypassing PaX ASLR protection. Phrack, Volume 0x0b, Issue 0x3b, Phile 0x09.

[3] Immunity Inc. DEPLIB.

[4] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). CCS 2007, pages 552-561.

Interpreter Specific

[5] Erik Cabetas. Vulnerabilities in Application Interpreters and Runtimes. SOURCE Boston 2009.

[6] Justin Ferguson. Advances in Attacking Interpreted Languages: Javascript. BA-Con 2008.

[7] Aaron Portnoy & Ali Rizvi-Santiago. Reverse Engineering Dynamic Languages, a Focus on Python. BA-Con 2008.

Shellcode

[8] Sinan Eran (noir). http://www.securityfocus.com/archive/82/327100/2009-02-24/1

Mitigations

[9] Microsoft. http://blogs.technet.com/srd/archive/2009/10/27/announcing-the-release-of-the-enhanced-mitigation-evaluation-toolkit.aspx

[10] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. Nozzle: A Defense Against Heap-spraying Code Injection Attacks. MSR-TR-2008-176.

Miscellaneous

[11] Tamarin Project. http://www.mozilla.org/projects/tamarin/faq.html

Thanks

A ton of people helped out. Silvio, guerilla, bla, tgo, _y_, lin0xx, wtbw, ens, wishi, mayhem, etc. are all much smarter than me. Dino Dai Zovi told me I should submit this as a talk and gave me initial feedback. Jon Petro read a draft and helped me fix a ton of mistakes. Manish Karir and Eric Wustrow for their discussions of possible implementation changes to mitigate these techniques. Matt Miller at MSEC and Peleus Uhley at Adobe provided pleasant vendor relations regarding this research. There are a few others I've spoken with so, if I left you out, I wasn't sure you'd want to named :p