Project Zero

News and updates from the Project Zero team at Google

Wednesday, August 19, 2015

Three bypasses and a fix for one of Flash's Vector.<*> mitigations

Posted by Chris Evans, Cookie Monster

With the release of Flash 18.0.0.209, two mitigations were introduced to combat abuse of `Vector` corruptions -- we covered these in a previous blog post. Flash 18.0.0.232 has just been released and it includes a change to the way one of the mitigations is implemented, to address Project Zero bug 482.

This blog post notes some ways to bypass the way Adobe implemented the `Vector.<*>` length checking mitigation. They are already fixed. It's not uncommon for new mitigations to go through some iterations to strengthen them after initial release and scrutiny. This blog post will cover the way the mitigation used to work, three possible bypasses of this mitigation, and the fix deployed.

How did the Vector.<*> mitigation work?
The mitigation worked by using a secret cookie to XOR into a copy of the length field. This XOR'ed value should not be guessable by the attacker, and it is checked whenever the length is used. Therefore, length corruptions should be trapped reliably at runtime.

All of the bypasses stem from where the secret cookie used to be stored. Let's dig through a running Chrome Linux x64 process to show how the secret cookie used to be stored. For convenience, we allocate a 512MB `Vector.<uint>` object. It's sufficiently large that it will get its own mapping, making it easy to find in `/proc/<pid>/maps`:

```
16748d5a8000-1674ad8a8000 rw-p 00000000 00:00 0
```

We also set the first value in the `Vector.<uint>` to be `0xf2f2f2f2`, meaning we can search memory to find the header that precedes the data:

```
(gdb) find/w 0x16748d5a8000,0x1674ad8a8000,0xf2f2f2f2
0x16748d6a8018
```

So now we can dump the object's header and first couple of data entries:

```
(gdb) x/8xw 0x16748d6a8000
0x16748d6a8000: 0x07ffff83 0x07ffff83 0xfd0c714d 0x00000000
0x16748d6a8010: 0xda146000 0x000007d7 0xf2f2f2f2 0x00000000
```

`0x7ffff83` appears twice, and represents the length and the capacity of the Vector. The third 32-bit value is the length XOR'ed with the secret cookie. We can calculate the value of the secret cookie with an additional XOR operation on the length and the XOR'ed length:

```
(gdb) p/x 0x07ffff83 ^ 0xfd0c714d
$4 = 0xfaf38ece
```

It's worth noting that the ability to calculate the cookie in this way isn't considered a "bypass". The mitigation is attempting to prevent a blind buffer overflow from being turned into a read primitive (and thus bypassing ASLR). If the attacker already has a read primitive and can read the length and XOR'ed length values, then they already have an ASLR defeat.

But -- here is the interesting part -- the secret cookie that is checked against is stored inside a structure that is pointed to by a pointer at offset `0x10` of the header we dumped out above. We can demonstrate that like this:

```
(gdb) find/w 0x7d7da146000,0x7d7da147000,0xfaf38ece
0x7d7da146c68
```

We see that the secret cookie is stored at offset `0xc68` into a large structure.

What are the three bypasses?
All three bypasses stem from the fact that the secret cookie value is fetched from a pointer in the `Vector` header. The attacker that is trying to corrupt a `Vector` length can of course corrupt the entire `Vector` header. With the same buffer overflow, it is possible to corrupt all of these things at the same time:

- The length of the `Vector` (`len`).
- The length of the `Vector` XOR'ed with the secret cookie (`lenXOR`).

Search This Blog

Search

Labels

- antivirus

Archives

- ► 2016 (10)
- ▼ 2015 (33)
  - ► December (2)
  - ► November (2)
  - ► October (1)
  - ► September (4)
  - ▼ August (6)
    - Windows 10^H^H Symbolic Link Mitigations
    - One font vulnerability to rule them all #4: Window...
    - Three bypasses and a fix for one of Flash's Vector...
    - Attacking ECMAScript Engines with Redefinition
    - One font vulnerability to rule them all #3: Window...
    - One font vulnerability to rule them all #2: Adobe ...
  - ► July (5)
  - ► June (4)
  - ► May (1)
  - ► April (1)
  - ► March (2)
  - ► February (3)
  - ► January (2)
- ► 2014 (11)

- The pointer from which the secret cookie is fetched (`ptr`).

The validation employed by the mitigation is effectively:

```
assert(len ^ lenXOR == ptr->cookie)
```

Unfortunately, a buffer overflow gives the attacker control of all the variables in the above validation, enabling at least a theoretical bypass. To bypass the mitigation successfully, the attacker has to forge a value of `ptr` which points to predictable content in order to gain control of the secret cookie value. Of course, with good ASLR, knowing where something is in memory is supposed to be hard. The bypasses cover various different tricks to defeat ASLR in order to reliably forge a secret cookie value.

Bypass #1: Heap spraying
This is the least sophisticated bypass because it relies on allocating large amounts of memory and is also only realistically feasible in a 32-bit address space. Above we used the phrase “good ASLR” and it is debateable whether “good ASLR” is possible at all in the browser context for 32-bit processes, but we include this bypass for completeness and because in practice it is practical.

The bypass is fairly simple: allocate e.g. 1GB of memory, perhaps with the `ByteArray` class. This will result in a 1GB contiguous allocation filled with the NUL byte. It is not possible to guess the start address of the allocation, but due to restrictions in the 32-bit address space, it is possible to very reliably guess an address that will mostly always be *somewhere within* the allocation. We can overwrite `ptr` above with the guessed address, leading to a `ptr->cookie` value of zero. With a known cookie value, we have defeated the mitigation.

Bypass #2: User Shared Data and vsyscall
Of course, a good bypass will work on 64-bit systems. This second bypass was first demonstrated on Windows 8.1 x64. It relies on a quirk of Windows ASLR where there unfortunately exists a mapping called **User Shared Data** that is at a fixed address in Windows processes. This mapping has a long history of abuse, including in a Pwnium 2 use-after-free exploit. In older Windows operating systems, User Shared Data contained function pointers, making it very dangerous to have at a fixed address. In Windows 8.1, there is only read-only data and no pointers, but it is still a violation of good ASLR to have any mapping at a fixed address.

To abuse User Shared Data to bypass the cookie, we simply need to observe that it is one page of data, and is zero-filled at the higher addresses in the page. Therefore, by setting `ptr` to be User Shared Data (`0x7ffe0000`), `ptr->cookie` is zero and the secret validation value we need for a given length becomes the same as the length.

To visualize how a buffer overflow would really go about this bypass, we can look at how the Vector header in memory would look both before and after a buffer overflow. Before:

```
        0x07ffff83      0x07ffff83      0x8b6fd5d7      0x00000000
        0x86f53000      0x00003d3c      0xf2f2f2f2      0x00000000
```

And after, with the bytes modified by the linear buffer overflow colored in red:

```
        0x07ffff84      0x07ffff84      0x07ffff84      0x00000000
        0x7ffe0000      0x00000000      0xf2f2f2f2      0x00000000
```

As can be seen, we’ve bumped the length up by one, set the validation value to be the same as the new length, and set `ptr` to 0x7ffe0000 so that the secret cookie is zero. This is a fairly powerful bypass as it applies to any linear buffer overflow where the attacker can write NUL bytes and can write at least 24 bytes into the `Vector` header. If the overflow writes more than 24 bytes into the header, that’s fine because it would proceed to harmlessly corrupt the `Vector`’s data.

In my opinion, Linux generally has stronger ASLR than Windows, but unfortunately on Linux there sometimes exists a mapping called **vsyscall**. And you guessed it -- it’s at a fixed address:

```
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

The higher parts of the page are filled with `0xcc` (int 3), including at the structure offset that the secret cookie is loaded from:

```
(gdb) x/16xw 0xffffffffff600c68
0xffffffffff600c68: 0xcccccccc 0xcccccccc 0xcccccccc 0xcccccccc
0xffffffffff600c78: 0xcccccccc 0xcccccccc 0xcccccccc 0xcccccccc
0xffffffffff600c88: 0xcccccccc 0xcccccccc 0xcccccccc 0xcccccccc
0xffffffffff600c98: 0xcccccccc 0xcccccccc 0xcccccccc 0xcccccccc
```

So, if we want a length of `0x07ffff84`, we can calculate the validation secret resulting from a secret cookie of `0xcccccccc`, and then write memory to simulate the linear buffer overflow leading to a bypass:

```
(gdb) p/x 0x07ffff84 ^ 0xcccccccc
$3 = 0xcb333348

(gdb) set *(unsigned int*)0x16748d6a8000 = 0x07ffff84
(gdb) set *(unsigned int*)0x16748d6a8004 = 0x07ffff84
(gdb) set *(unsigned int*)0x16748d6a8008 = 0xcb333348
```

```
(gdb) set *(unsigned int*)0x16748d6a8010 = 0xff600000
(gdb) set *(unsigned int*)0x16748d6a8014 = 0xffffffff

(gdb) x/8xw 0x16748d6a8000
0x16748d6a8000: 0x07ffff84 0x07ffff84 0xcb333348 0x00000000
0x16748d6a8010: 0xff600000 0xffffffff 0xf2f2f2f2 0x00000000
```

Fortunately, at least vsyscall on Linux is deprecated. You can already turn it off with likely no ill effects with the boot parameter `vsyscall=none`. Hopefully the next generations of Linux distributions will turn it off by default.

Bypass #3: Partial pointer overwrite
So, can we bypass the mitigation on 64-bit Linux with the `vsyscall=none` boot parameter. It turns out that we can! We're going to use a technique called a "partial pointer overwrite". First, let's look at the memory content immediately after the secret cookie value:

```
(gdb) x/16xw 0x7d7da146c68
0x7d7da146c68: 0xfaf38ece 0x00000000 0x00000000 0x00000000
0x7d7da146c78: 0x00000000 0x00000000 0x00000000 0x00000000
0x7d7da146c88: 0x00000000 0x00000000 0x00000000 0x00000000
0x7d7da146c98: 0x00000000 0x00000000 0x00000000 0x00000000
```

It looks like the secret cookie value might be the last item in a large structure. It's not clear whether we can rely on zero initialization of the fields, but for now we can demonstrate the issue using these zero values. (An alternative would have been to find a reliably zero value earlier in the structure.) The other observation we make is that the structure that stored the secret cookie is sufficiently large that it gets allocated at a page boundary. Putting it all together, we can simulate a linear buffer overflow bypassing the mitigation like this:

```
(gdb) set *(unsigned int*)0x16748d6a8000 = 0x07ffff84
(gdb) set *(unsigned int*)0x16748d6a8004 = 0x07ffff84
(gdb) set *(unsigned int*)0x16748d6a8008 = 0x07ffff84
(gdb) set *(unsigned char*)0x16748d6a8010 = 0x04
```

The effect of the final write above is to overwrite the least significant byte of `ptr` with `0x4`. Since the structure pointed to is page aligned, we effectively increase the value of ptr by `0x4`, which causes the secret cookie to become zero.

Although the partial pointer overwrite works without needing any ASLR quirks or weaknesses, it does require that that attacker's buffer overflow has perfect byte-by-byte control. Buffer overflows that write in multiples of (e.g.) 32-bit quantities will not be suitable.

Fixing the mitigation
Adobe fixed the mitigation by moving the storage of the secret cookie from `ptr` into the static BSS. Therefore, corrupting `ptr` will no longer enable the attacker to change the program's view of what the secret cookie is.

In addition, Adobe have now enabled the `Vector` heap partitioning mitigation for all platforms. `Vector` heap partitioning can be a useful defense-in-depth for attacks, including some of the attacks described here.

Posted by Unknown at 1:10 PM G+1 +16 Recommend this on Google

No comments:

Post a Comment

Enter your comment...

Comment as: Google Account

Publish Preview

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)