19 April, 2013

Share

Recently, MWR Labs took part in the Pwn2Own 2013 competition in Vancouver, demonstrating a full sand-box bypass exploit against Google Chrome. The exploit used two vulnerabilities:

- A type confusion in WebKit, Chrome's rendering engine at the time (CVE-2013-0912)
- A kernel pool overflow in Microsoft Windows, the underlying operating system

This blog post will provide in-depth technical details of the first vulnerability, with a second blog post upcoming to detail the second vulnerability.

The Vulnerability

The vulnerability in WebKit (2) was a type confusion vulnerability in the handling of view targets in SVG documents. When loading an SVG document into a web page, it is possible to specify a viewTarget for the document, which specifies a target for the current view. The vulnerable code from WebCore/svg/SVGViewSpec.cpp that was used to retrieve this property of an SVG document is shown below.

```
SVGElement* SVGViewSpec::viewTarget()
const
{
if (!m_contextElement)
```

```
return 0;
return static_cast<SVGElement*>
(m_contextElement->treeScope()-
>getElementById(m_viewTargetString));
}
```

m_viewTargetString in this context is the string assigned to the viewTarget property of the SVG document. The code gets the element with the corresponding id from the SVG document, casts it to an SVGElement, and returns a handle to JavaScript. The assumption is that the element returned from the call to getElementById() will be an SVG element. However, it is possible to embed non-SVG elements inside an SVG document using the foreignObject tag. Setting the SVG document's viewTarget property to an id of a non-SVG element inside a foreignObject tag, and then retrieving the viewTarget property of the SVG document in JavaScript caused the non-SVG element to be cast to an SVGElement, and returned a handle to this element to JavaScript.

Exploitation

This vulnerability was extremely flexible, as we were able to cast an element into almost any SVG element type. We did this by creating an HTML element with the tag name set as a valid SVG tag. In the context of HTML, this tag name is invalid, so when the page is rendered, an HTMLUnknownElement is constructed. After the cast, the unknown element is recog-

nised as a valid SVG element, and the properties and methods of this SVG element can be accessed from JavaScript. Since an HTMLUnknownElement object always has a smaller memory footprint than a valid SVG object, accessing SVG-specific attributes of the object causes the adjacent memory to be interpreted as part of the SVG object.

By setting up the heap correctly, we were able to control the adjacent memory contents, and control the attributes of the SVG object. Because we could read the adjacent memory without the risk of crashing the browser, we were able to detect when we controlled the adjacent memory, and could therefore avoid the need for excessive allocations on the heap. During testing, we observed that it usually took around 10 allocations before the desired heap layout was achieved, which is faster and more efficient than traditional heap spraying.

We were also able to exploit this vulnerability multiple times without crashing the browser, each time setting up the heap and triggering the invalid cast to a different SVG element with the most advantageous object layout for the desired task. We used five seperate exploit stages in the final exploit, which are detailed below.

1. Leaking a pointer

As with many modern applications, Chrome makes full use of Address Space Layout

Randomisation (ASLR), and consequently we are unable to reliably determine where key objects are in memory in order to corrupt them. An important step in the exploit was to determine the base address of a dll in memory, so that this could be used to construct a ROP chain later on in the exploit. The first step for us to do this was to leak a pointer from within chrome.dll.

We did this by filling the adjacent memory of our “confused” object with HTML div elements. We were able to read the vtable pointer of the adjacent object by reading a property of the SVG element, but we were unable to verify that this was read from a valid object with this information alone. Rather than guessing and risking the reliability of the exploit, we took additional steps to confirm that the object we were accessing was indeed one of the div elements we allocated previously.

Another property we were able to read from the SVG element corresponded to the lastChild pointer of a div element, assuming that the heap was laid out correctly. We kept a list of references to all div elements allocated when setting up this stage of the exploit, and accessed what we believed was the lastChild property of this element. If the value was null, this indicated that the div had no children, which was expected, as we created the divs and did not assign them any children. We then assigned a child to each div and monitored for a change in the lastChild of the adjacent object. If the value changed, this indicated that the object was one of our allocated div elements, and

we could safely assume the vtable pointer read from that object was valid. As a bonus, the leaked lastChild value is actually a pointer to the child object, which we use later on in the exploit to hold our ROP chain.

2. Calculating the base address

We could have subtracted a static value from the leaked pointer to find the base address of chrome.dll, but this has several downsides, such as making the exploit dependant on the exact version of chrome being targeted, as this is likely to change often (an excellent example being Chrome's last-minute patches released 2 days before the contest). Instead, we opted to calculate the value dynamically.

We were able to do this by using an object which used vectors (arrays) as part of it's structure that were accessible from JavaScript. In memory, vectors have a base address and a capacity assigned. These values are used to determine the boundaries of the vector in memory. We filled the adjacent memory with strings that we controlled, which allowed us to specify an arbitrary value for the base address and size of the vector. We placed the base address well before chrome.dll was expected to begin, and made the size large enough to cover from there to the leaked pointer (we know the beginning of the dll cannot be after this point). We then used standard vector functions to read backwards through memory in page-sized chunks, looking for the "MZ" signature in the

PE header of chrome.dll. This gives us the base address of the image in memory.

3. Reading chrome.dll's .text segment into memory

Again, we could have built a ROP chain that used static offsets from the base address we calculated previously, but this is subject to change with each minor version of Chrome. We decided that the best option would be to dynamically build the ROP chain by reading the contents of chrome.dll's text segment into a JavaScript string, and then use the native string functions to find sequences of bytes in that string.

To read the memory, we used the same vector manipulation techniques we used when calculating the base address to place a vector over the .text segment, and read the bytes using standard vector functions. The vector used in this stage was a vector of float values, so the memory read from the dll had to be converted from a float representation to a series of hexadecimal bytes. We did this by initialising an ArrayBuffer, and creating two views over it, one as a Float32Array, and one as a Uint32Array. We were then able to assign values to the Float32Array and read them out as bytes from the Uint32Array. Some values did not decode cleanly as valid floats, but these could be safely ignored, as the dll was adequately sized to contain a majority of gadgets.

4. Building the ROP chain

The native string functions used to find gadgets in the string were extremely fast. The string contained almost 40 megabytes of data, and yet when benchmarking the gadget finding function, a search for 100000 gadgets took under a second. We now wanted to place our ROP chain at a known address in memory. To do this, we made use of the address of child element we assigned to the div in stage 1. We created a vector that covers this child element, and read an offset from the vector to find the address of a string property. Again, we were able to confirm that this was leaked from the correct object by assigning a value to the property and observing a change.

After building the ROP chain, we assigned it to the string property, and stored the leaked address of the ROP chain for use in the final stage of the exploit.

5. Gaining code execution

To gain control of the program flow, we used the simplest method available, allocating controlled strings adjacent to the object and calling a virtual method on the object, which uses the controlled memory as a vtable pointer. We used this to execute a series of stack pivot gadgets, and to begin execution of our ROP chain.

Once in the ROP chain, we allocated a page of RWX memory, and copied in the bytes for the remaining gadgets we wanted to execute, as

well as the shellcode to trigger the kernel vulnerability.

Sandbox Bypass

During the source code review, we also identified a vulnerability (3) in Chrome's IPC layer, that could have been used as part of a sandbox bypass. However, this bug was discovered independently by Google, and patched prior to the contest. In parallel, we were auditing the Windows kernel in order to bypass the sandbox entirely and gain code execution as SYSTEM. This process will be documented in the second part of this blog post, once the vulnerability fix has been released by the vendor.

Conclusion

It is very difficult to secure such a complex piece of software, which frequently deals with untrusted input. Even with modern exploit mitigation techniques and the inclusion of a sandboxed renderer process, these protection mechanisms can be circumvented by exploiting the underlying operating system, which may not be so well protected. Several attempts are being made to improve the security of browsers in the future, including Mozilla's rendering engine, Servo (4), which is written in the managed language Rust. These approaches introduce their own challenges, such as matching the performance of purely native implentations.

References

(1) http://labs.mwrinfosecurity.com/blog/2013/03/06/pwn2own-at-cansecwest-2013/
(2) https://trac.webkit.org/changeset/145013/
(3) https://code.google.com/p/chromium/issues/detail?id=169770
(4) http://www.mozilla.org/en-US/research/projects/