Taming wild copies
From hopeless crash to working exploit
Taming wild copies
From hopeless crash to working exploit
Who am I?

- Chris Evans / scarybeasts / Troublemaker
- Started Project Zero [current]
- Started Chrome Security Team
- Security research
- vsftpd / privsep
- Certificate pinning / PartitionAlloc
Overview

1. Introduction to Project Zero
2. Reviewing historic wild copies
3. Exploiting a new wild copy in a new way
4. Conclusions / Q & A
Overview

1. Introduction to Project Zero
2. Reviewing historic wild copies
3. Exploiting a new wild copy in a new way
4. Conclusions / Q & A
Project Zero

- Covered in more detail in the earlier keynote.
- Announced July 2014, mission is to “Make 0-day hard”.
- Providing desirable jobs for the best offensive security researchers in the industry.
- Bringing top-tier offensive research back into the open.
- Follow our blog:
  [http://googleprojectzero.blogspot.com/](http://googleprojectzero.blogspot.com/)
Project Zero and exploitation

- We only exploit a small percentage of vulnerabilities we find.
- When does it make sense?
  - To advance the public state of the art.
  - To generally counter “not exploitable” arguments.
  - To allow high-quality real offense to guide defense.
  - To contribute to the public body of exploitation data.
Overview

1. Introduction to Project Zero
2. Reviewing historic wild copies
3. Exploiting a new wild copy in a new way
4. Conclusions / Q & A
“Wild copy”?  

We define a “wild copy” as one where:

- The attacker has limited control over the length of the copy.
- The copy is sufficiently large that it will always fault before completion.
  - Think about length “-1”.

Historic wild copy: eglibc memcpy()

- A vulnerability *inside* (e)glibc memcpy().
- **Disclosed** mid-2011.
- **Independent discovery** in August 2011 by Chromium Vulnerability Rewards program participant Aki Helin.
- Negative length causes incorrect use of out-of-bounds jump table entry…!
  - Auto-defeats ASLR, DEP.
- Attacker needs precise control of length.
Historic wild copy: Java ICC profiles

- An interesting memcpy(..., ..., -1) bug in the Java Runtime Environment, from 2006.
- Affecting JPEG parsing via ICC profiles.
- The JRE has a very complicated SIGSEGV handler installed.
  - Generates verbose hs_err_pidXXXXX.log file.

Java frames: (J=compiled Java code, j=interpreted, Vv=VM code)
- ~RuntimeStub::_complete_monitor_locking_Java
- J org.apache.mina.transport.socket.nio.SocketIoProcessor.doFlush(Lorg/apache/mina/transport/socket/nio/SocketSessionImplV
- [...]
- Java Threads: ( => current thread )
- =>0x08ddb400 JavaThread "SocketAcceptorIoProcessor-1.2" [thread_in_vm, id=15329, stack(0x63a5b000,0x63aac000)]
- [...]
- Heap
  - PSYoungGen total 3712K, used 875K [0x9ab10000, 0x9afc0000, 0xb3b10000]

- SIGSEGV re-raised inside handler.
Historic wild copy: Apache chunked encoding

- A negative memcpy bug from 2002 ([advisory](#)) ([more details](#)).
- Initially declared unexploitable.
Historic wild copy: Apache chunked encoding

- A negative memcpy bug from 2002 ([advisory](#)) ([more details](#)).
- Initially declared unexploitable.
Historic wild copy: Apache chunked encoding

- A negative memcpy bug from 2002 ([advisory](#)) ([more details](#)).
- Initially declared unexploitable.
- A copy from the heap to the stack.
- FreeBSD memcpy() had memmove() semantics:
  - Backwards copy
  - Copies remainder first.
  - **Reloads length from stack** after remainder copy.
Historic wild copy: Apache chunked encoding

```
dst_buf
```

```
ap_bread()
arg: len
dst_buf + 0xffffffff6e or dst_buf - 146
arg: src
arg: dst
```

```
cmemcp()y
```
Historic wild copy: Apache chunked encoding

```
0xbffff000
```

```
dst_buf
```

```
ap_bread()
arg: len
arg: src
arg: dst
```

```
dst_buf + 0xffffffff6e or dst_buf - 146
```

```
copy()
```
Historic wild copy: Apache chunked encoding

\[
\text{dst}_\text{buf} \\
0xbfffff000
\]

\[
\text{ap}_\text{bread}() \quad \text{dst}_\text{buf} + 0xffffffff6e \quad \text{or} \quad \text{dst}_\text{buf} - 146
\]

\[
\text{memcpy}() \\
\text{arg: len} \\
\text{arg: src} \\
\text{arg: dst}
\]
Overview

1. Introduction to Project Zero
2. Reviewing historic wild copies
3. Exploiting a new wild copy in a new way
4. Conclusions / Q & A
New wild copy: Flash G711

Introducing the bug:

- Project Zero [issue 122](#).
- Found by a Chris Evans / Tavis Ormandy fuzzing collaboration.
- Fixed by Adobe in the [November 2014 Flash patch](#).
- G711 is an audio codec from, uh, 1972.
- Likely disused in Flash.
  - On some platforms (including Internet Explorer), any attempt to use the codec will crash.
- Crash will always be memmove(heap - 2048, heap, -2048).
New wild copy: exploitation environment

- I don’t always write exploits but when I do.....
  - Linux x64 (Chrome browser if applicable)
    - Best ASLR?

- Problem: for random reasons, Chrome Flash never had this bug.

- libpepflashplayer.so binary patch time!

Old:
```
0x41 0xbc 0xe 0x05 0x00 0x00    mov    $0x51e,%r12d
```

New:
```
0x41 0xbc 0xde 0x03 0x00 0x00    mov    $0x3de,%r12d
```

- Sandbox not tackled.
New wild copy: exploitation environment

- SIGSEGV handler is installed.
- See ExceptionHandler::SignalHandler

```cpp
std::vector<ExceptionHandler*>* g_handler_stack_ = NULL;
pthread_mutex_t g_handler_stack_mutex_ = PTHREAD_MUTEX_INITIALIZER;

void ExceptionHandler::SignalHandler(int sig, siginfo_t* info, void* uc)
{
    // All the exception signals are blocked at this point.
    pthread_mutex_lock(&g_handler_stack_mutex_);
    [...]  
    for (int i = g_handler_stack_->size() - 1; !handled && i >= 0; --i)  
    {
        handled = (*g_handler_stack_)[i]->HandleSignal(sig, info, uc);
    }
```

- Both BSS and heap touched.
New wild copy: how?

- How long do we have?
  My machines (laptop / desktop) copy at \(~10\text{GB} / \text{sec}\). Virtual address space limited to 4GB. This gives us \(~0.4\text{s}\).

  More realistically, an exploit might not want to use \(>100\text{MB}\). This gives us \(~0.01\text{s}\).

  “Safari hacked in 5 seconds at Pwn2Own” -- The Loop

- How will we gain control?
  - The corruption occurs on the audio thread.
  - The script threads run in parallel.
  - Let’s observe and abuse a side effect via the script thread!
  - “Parallel Thread Corruption”
New wild copy: exploit plan

- **Let’s corrupt a** `Vector.<uint>`
  - This is the current go-to method for Flash exploitation.
  - For example: Project Zero [regex exploit](#).
  - And: Project Zero [4GB-out-of-bounds exploit](#).
  - Also: [in the wild](#).

- **Why are** `Vector.<uint>` **corruptions so powerful?**

```
(gdb) x/12xw 0x7fd6ce038000
0x7fd6ce038000: 0x000007d0 (0x00000000) 0x12354000 0x00007fd7
0x7fd6ce038010: 0xfeeldead 0x00000000 0x00000000 0x00000000
0x7fd6ce038020: 0x00000000 0x00000000 0x00000000 0x00000000
```

- **Size is fully under attacker control.**
New wild copy: exploit plan

- Let’s groom the heap

G711 object

Vector. <uint>

Runway: 100MB

0xffe1dead
New wild copy: exploit plan

- Let’s groom the heap

G711 object

Vector. 
< unint >

Runway: 100MB

0xfee1dead
New wild copy: exploit plan

- Let’s groom the heap

G711 object

Vector. <uint>

0xffee1dead
New wild copy: exploit plan

- Let's read a `Vector.<Object>`
Understanding the Flash heap

- To heap groom, we need to understand the heap.
- Great news: the Flash heap is open source.
- Heap properties:
  - Block based (1 block == 4KB page)
  - Free block coalescing
  - Bucket based (e.g. 1 block might be set aside for allocations from 121 - 128 bytes)
  - Extent based (16MB mappings on 64-bit, committed in at least 128KB chunks)
  - VirtualAlloc() / mmap() based
  - Inline metadata without guard pages
  - Falls back to raw, unhinted VirtualAlloc() / mmap() for > 1MB
  - Not particularly hardened
    - Not a huge deal IMHO
Understanding the Flash heap

- But this heap has quirks!
- Annoy ing ones.
- The most relevant quirks:
  - The heap tries to extend forwards but typically extends backwards.
    - Applies 100% to Linux x64, maybe other platforms.
    - An interaction between the heap, a mapping collision, and the default OS allocation strategy.
    - Ends up creating “accidental” guard pages.

```
7fd6e7985000-7fd6e87c5000  rw-p 00000000 00:00 0 // 1.8MB
7fd6e87c5000-7fd6e8985000  ---p 00000000 00:00 0
7fd6e8985000-7fd6e9765000  rw-p 00000000 00:00 0 // 2.2MB
7fd6e9765000-7fd6e9985000  ---p 00000000 00:00 0
7fd6e9985000-7fd6ea725000  rw-p 00000000 00:00 0 // 2.5MB
7fd6ea725000-7fd6ea985000  ---p 00000000 00:00 0
7fd6ea985000-7fd6eb6c5000  rw-p 00000000 00:00 0 // 2.9MB
7fd6eb6c5000-7fd6eb985000  ---p 00000000 00:00 0
```
“Accidental guard pages?!”
Understanding the Flash heap

- The most relevant quirks:
  - Inline heap metadata is extended as the heap grows.
    - Creates free holes in your heap as you spray it.
  - Free block re-use is not MRU.
    - See GCHep::addToFreeList.
  - JIT pages are intermingled.

- Random non-security quirks:
  - Heap expansion appears to be O(n^2).
    - See GCHep::ExpandHeapInternal.
  - free() looks O(n) for large allocation sizes.
    - See GCHep::AddrToRegion.
  - Some sizes are space inefficient, e.g. 4097 bytes -> 8192 allocation.
Grooming the Flash heap

1. Allocate a large allocation (e.g. 1GB):

   [ ---- 1GB ---- ] [ heap: used | reserved ]

2. Spray block-sized allocations to force a new extent

   [ heap: used | reserved ] [ ---- 1GB ---- ] [ heap: used ]

3. Free the large allocation:

   [ heap: used | reserved ] [ free! ] [ heap: used ]
Grooming the Flash heap

Putting our ducks in a row:

- 8KB ByteArray buffer
- 8KB ByteArray buffer
- 8KB ByteArray buffer
- Vector. <uint> (2000)
- Vector <Object> (1000)
- Landing runway (100MB)
Grooming the Flash heap

Putting our ducks in a row:

- Must perform another heap spray (8KB chunks) before creating the heap hole, thanks to the non-MRU quirk.

| 8KB ByteArray buffer | Free 8KB chunk | 8KB ByteArray buffer | Vector. <uint> (2000) | Vector <Object> (1000) | Landing runway (100MB) |
Pulling the trigger

Putting our ducks in a row:

- 8KB ByteArray buffer
- Audio object
- 8KB ByteArray buffer
- Vector. <uint> (2000)
- Vector <Object> (1000)
- Landing runway (100MB)
ROP-free exploit in 4 memory accesses and a free()
ROP-free exploit in 4 memory accesses and a free()

1. Relative read: vtable
ROP-free exploit in 4 memory accesses and a free()

2. Relative read: Vector.<Object> location
   We can now read and write absolute memory locations.
ROP-free exploit in 4 memory accesses and a free()

3. Absolute read: `__memmove_ssse3_back` from `memmove@got.plt`
ROP-free exploit in 4 memory accesses and a free()

4. Absolute write: `__libc_system` to `munmap@got.plt`
ROP-free exploit in 4 memory accesses and a `free()`

5. Free a ByteArray buffer allocated earlier.

```
killall -STOP chrome;
gnome-calculator
```
ROP-free exploit in 4 memory accesses and a free()

6. Perform sums
Notes on reliability

- Exploit is surprisingly reliable given the nature of the bug.
- Reasons for exploit to fail:
  - OS scheduling; script thread gets descheduled while audio thread runs.
  - Heap has lots of free blocks; spray fails to fill them up, causing the audio object to be allocated after the 1GB object and not before it.
  - Other thread activity; threads other than our current thread interfere with our heap grooming at just the wrong time.
  - Heap is already very large; post 1GB-allocation spray creates two heap reservations, leading to accidental guard pages.
Mitigations

- Vector.<uint> and related objects are dangerous.
  - Apply heap partitioning.
- ByteArray buffers allow very fine-grained heap control.
  - Apply heap partitioning.
- Very large allocations appear at predictable addresses relative to the rest of the heap.
  - Randomize them.
- The heap is mapped within close proximity to the binaries on 64-bit platforms.
  - Randomize the heap start location better.
- Lack of RELRO.
  - Probably not a significant defense against this and related attacks.
Conclusions

- Avoid declaring any memory corruption unexploitable.
- Treat wild copies as likely exploitable within multi-threaded scripting environments.