Fundamentos de Segurança Informática (FSI)

2024/2025 - LEIC

Software Security (Part 2)

Hugo Pacheco
hpacheco@fc.up.pt

The infamous “buffer overflow” & friends

- One of the simplest (conceptually!) examples of gaining of control; belongs to a large class of memory management errors in (not only) C
- When the program “accesses outside” of its allocated memory

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2022
1	CWE-787	Out-of-bounds Write	63.72	70	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.54	4	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	34.27	6	0
4	CWE-416	Use After Free	16.71	44	+3
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	15.65	23	+1
6	CWE-20	Improper Input Validation	15.50	35	-2
7	CWE-125	Out-of-bounds Read	14.60	2	-2
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.11	16	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	11.73	0	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.41	5	0
11	CWE-862	Missing Authorization	6.90	0	+5
12	CWE-476	NULL Pointer Dereference	6.59	0	-1
13	CWE-287	Improper Authentication	6.39	10	+1
14	CWE-190	Integer Overflow or Wraparound	5.89	4	-1

[2023 CWE Top 25: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html]

Heap Buffer Overflows

Heap Buffer Overflows

- **Vulnerability**: essentially the same as a stack buffer overflow…

```
int main()
{
 char *buffer;
 buffer = (char*)malloc(5*sizeof(char));
 char *string;
 string = (char*)malloc(5*sizeof(char));
 strcpy(buffer,"abcdefghijklmnopqrstuvwxyz1234567890");
 printf("String: %s \n\n", string);
 return 0;
}
```

- Where does the copied string overflow into?

Heap Buffer Overflows

- Where does the copied string overflow into?

```
int main()
{
 char *buffer;
 buffer = (char*)malloc(5*sizeof(char));
 char *string;
 string = (char*)malloc(5*sizeof(char));
 strcpy(buffer,"abcdefghijklmnopqrstuvwxyz1234567890");
 printf("String: %s \n\n", string);
 return 0;
}
```

- C standard only guarantees blocks of contiguous memory

- Depends on dynamic memory allocation
  ⇒ varies across OS + HW architecture

- **Exploit**: much more complicated…

Heap Buffer Overflows

- When a *heap buffer overflow* occurs, usurping control is also possible
- Many candidate targets:
  - function pointers
    (e.g., virtual function tables in C++ and global offset tables for dynamically-linked libraries)
  - exception handlers
    (e.g. linked list of pointers to functions in C++)
  - `longjmp` buffers for
    exception handling in C
  - …

[https://www.cgsecurity.org/exploit/heaptut.txt]

```
w00w00!  For those of you that are saying, "Okay.  I see this works in a
controlled environment; but what about in the wild?"  There is sensitive
data on the heap that can be overflowed.  Examples include:
     functions                          reason
  1. *gets()/*printf(), *scanf()        __iob (FILE) structure in heap
  2. popen()                            __iob (FILE) structure in heap
  3. *dir() (readdir, seekdir, ...)     DIR entries (dir/heap buffers)
  4. atexit()                           static/global function pointers
  5. strdup()                           allocates dynamic data in the heap
  7. getenv()                           stored data on heap
  8. tmpnam()                           stored data on heap
  9. malloc()                           chain pointers
  10. rpc callback functions            function pointers
  11. windows callback functions        func pointers kept on heap
  12. signal handler pointers           function pointers (note: unix tracks
      in cygnus (gcc for win),          these in the kernel, not in the heap)

Now, you can definitely see some uses these functions.  Room allocated
for FILE structures in functions such as printf()'s, fget()'s,
readdir()'s, seekdir()'s, etc. can be manipulated (buffer or function
pointers).  atexit() has function pointers that will be called when the
program terminates.  strdup() can store strings (such as filenames or
passwords) on the heap.  malloc()'s own chain pointers (inside its pool)
can be manipulated to access memory it wasn't meant to be.  getenv()
stores data on the heap, which would allow us modify something such as
$HOME after it's initially checked.  svc/rpc registration functions
(librpc, libnsl, etc.) keep callback functions stored on the heap.
```

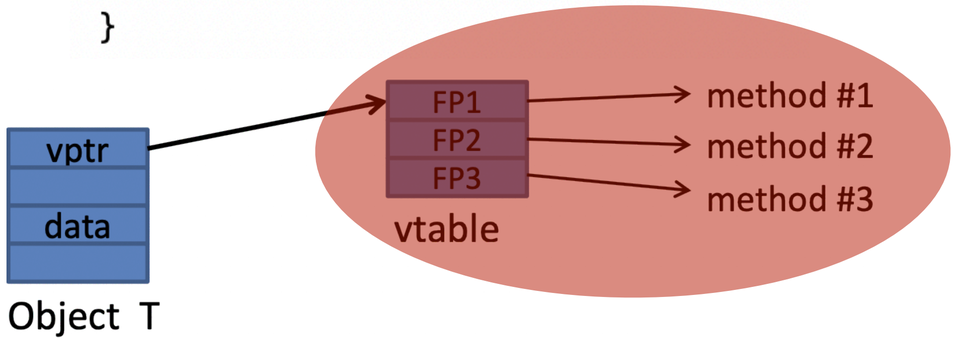
The **setjmp**() function saves various information about the calling environment (typically, the stack pointer, the instruction pointer, possibly the values of other registers and the signal mask) in the buffer *env* for later use by **longjmp**(). In this case, **setjmp**() returns **0**.

The **longjmp**() function uses the information saved in *env* to transfer control back to the point where **setjmp**() was called and to restore ("rewind") the stack to its state at the time of the **setjmp**() call. In addition, and depending on the implementation (see NOTES), the values of some other registers and the process signal mask may be restored to their state at the time of the **setjmp**() call.

vTables

- How do Object-Oriented languages handle method/ function calls? (e.g., C++)
- When `obj->foo()` is called by `bar()`, how is control passed to `foo()`?
- Object is stored in the *heap*, and it contains a **pointer to its *virtual function table*** (or *vTable*)
- The *heap* will have plenty of function pointers!

```
class Base
{ public: virtual void foo()
    {cout << "Hi\n";} };

class Derived: public Base
{ public: void foo()
    {cout << "Bye\n";} };

void bar(Base* obj)
{ obj->foo(); }

int main(int argc, char* argv[])
{
        Base *b = new Base();
        Derived *d = new Derived();

        bar(b);
        bar(d);
 }
```

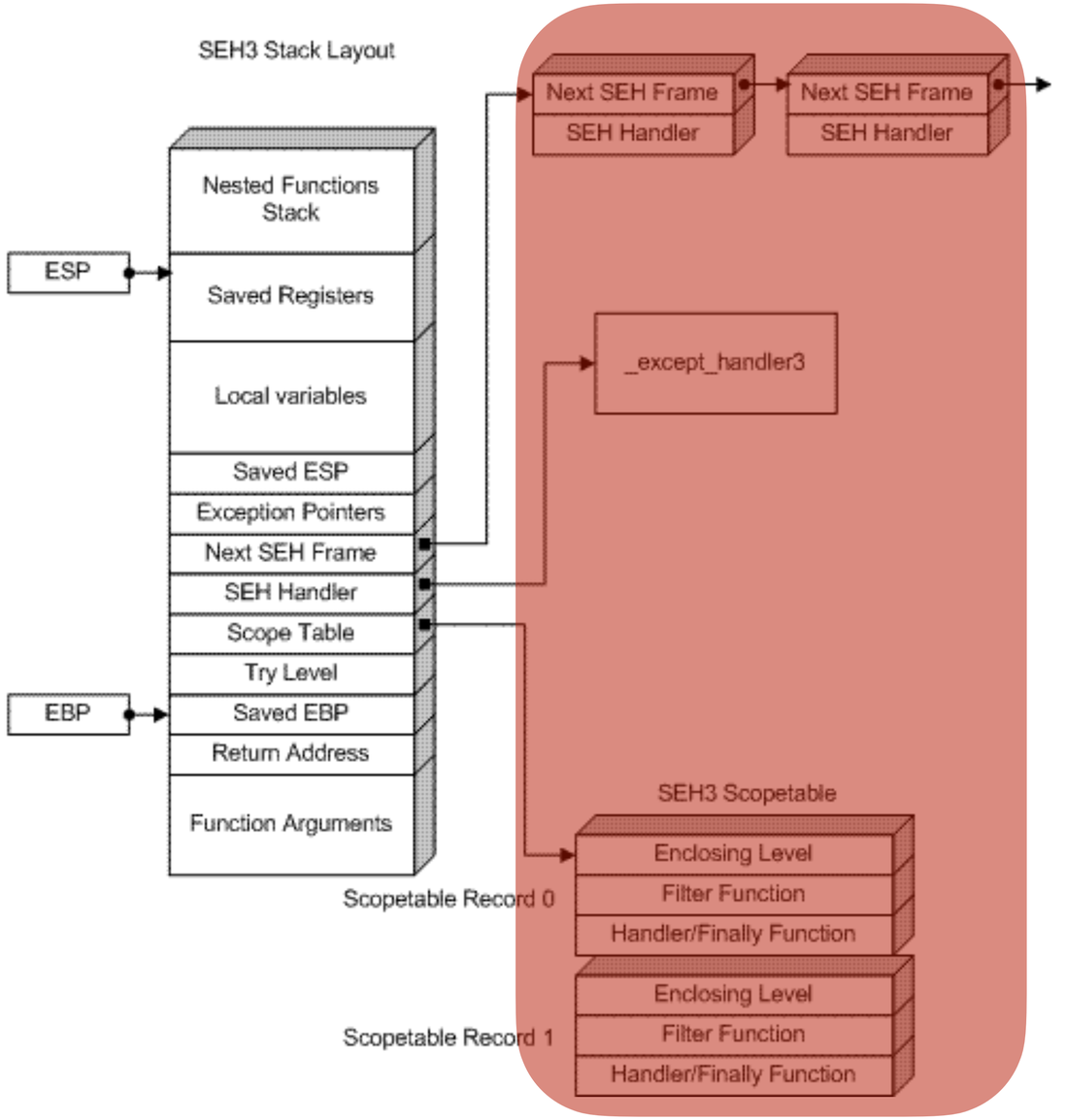


Exception Handlers

- In languages with exceptions (e.g., C++):
  - The stack frame of a function includes a linked list of **pointers to different exception handling routines**

1. heap buffer overflow ⇒ overwrite exception handling address

2. exception ⇒ execution proceeds to written address



malloc/free

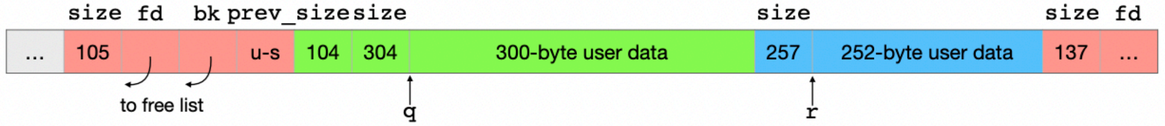
- Allocating/deallocating blocks/chunks of memory in the heap
- The dynamic memory allocator keeps a certain block control structure
  - `prev_size`: size of the previous chunk in memory
  - `size`: size of this chunk in use
  - `fd`: forward pointer in free list
  - `bk`: backward pointer in free list


```
void *p = malloc(100);
```

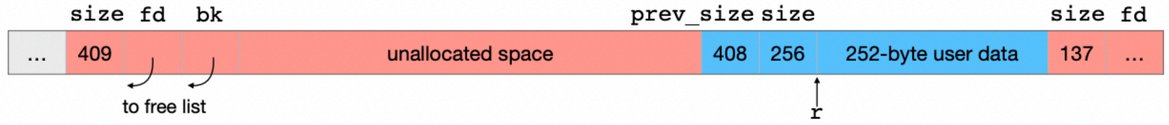

```
void *q = malloc(300);
```


```
free(p);
```

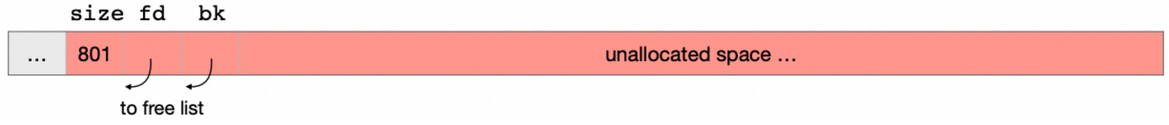

```
void *r = malloc(252);
```



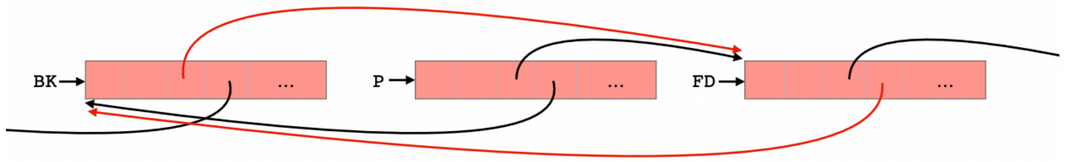

```
free(q);
```




```
free(r);
```



- When we call `free`:



malloc/free

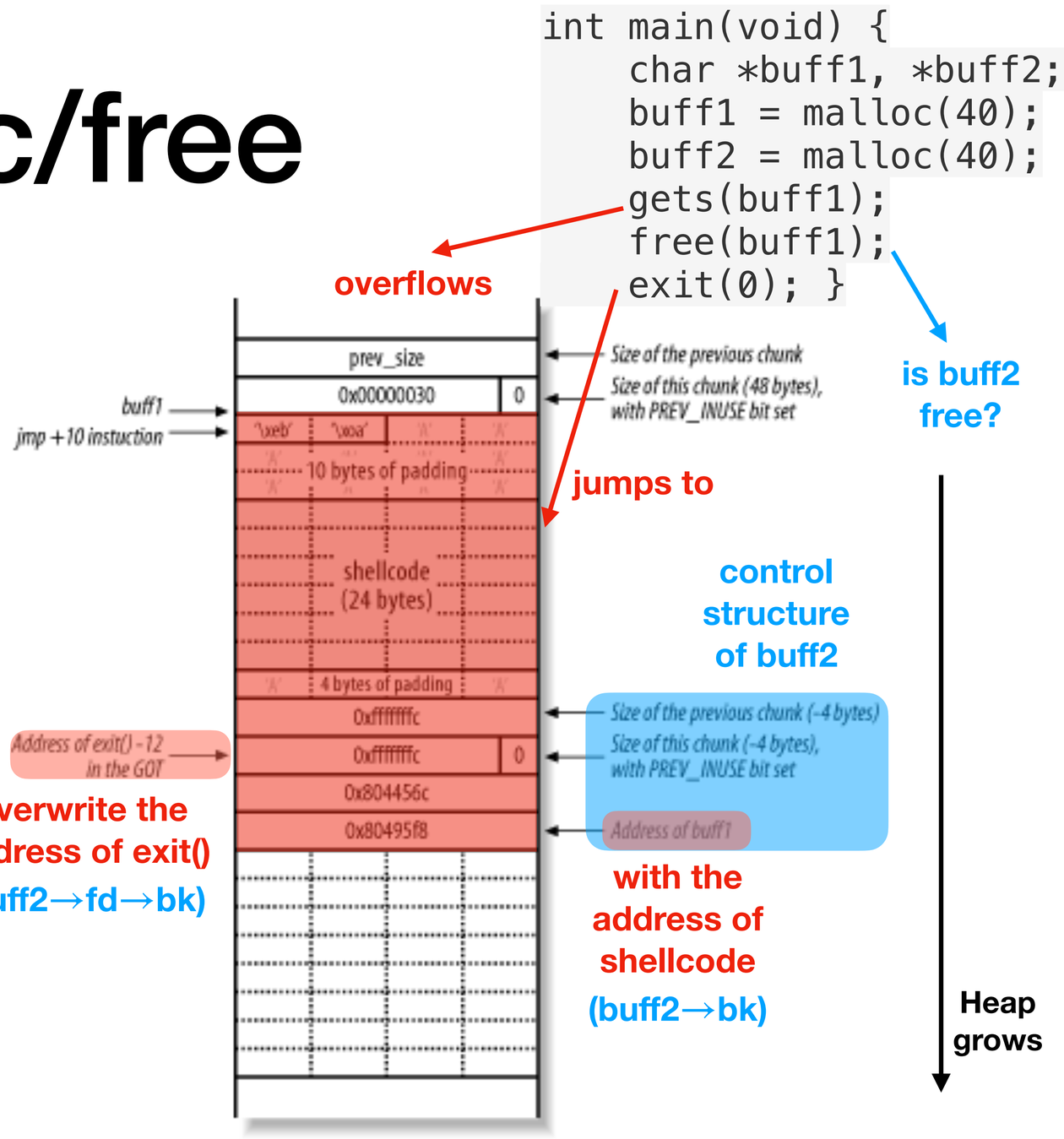
- “Smashing the heap” (memory allocator):

  1. write the “payload” to the heap memory
     ⇒ overwrite malloc/free block pointers

  2. `free` changes other memory locations
     ⇒ overwrite other function addresses

     - **deletes a block in a linked list, rewriting pointers** ⇒ if previous/next block is free, merge them

     - ends up **overwriting a function pointer with the shellcode address**



http://etutorials.org/Networking/network+security+assessment/Chapter+13.+Application-Level+Risks/13.5+Heap+Overflows/

Heap Overflow Examples

Overview

Microsoft Windows Media Player contains a buffer overflow vulnerability that may allow a remote, unauthenticated attacker to execute arbitrary code on a vulnerable system.

Description

Microsoft Windows Media Player (WMP) is an application that ships with Microsoft Windows systems used to play various types of media files. Windows Media Player fails to properly validate bitmap image files (.bmp), potentially allowing a buffer overflow to occur.

If the size field in the bitmap header is set to set to a value that is less than the actual size of the file, WMP will allocate an under-sized buffer to hold the bitmap. When data is copied to this buffer, the buffer overflow may occur. For more information, please see Microsoft Security Bulletin MS06-005.

```
Windows Media Player can play bit map format files, such as a .bmp file
and use Windows Media Player (WMP) to decode the .dll process bmp file.
But it can't correctly process a bmp file which declares it's size as 0.
In this case, WMP will allocate a heap size of 0 but in fact,  it will
copy to the heap with the real file length. So a special bmp file that
declares it's size as 0 will cause the overflow. When changing the size
```

> Buffer overflow in the exif_read_data function in PHP before 4.3.10 and PHP 5.x up to 5.0.2 allows remote attackers to execute arbitrary code via a long section name in an image file.

The bug was discovered 12/15/2004. The weakness was presented 01/10/2005 (Website). The advisory is shared at redhat.com. This vulnerability is uniquely identified as CVE-2004-1065 since 11/23/2004. The exploitability is told to be easy. It is possible to initiate the attack remotely. No form of authentication is needed for exploitation. Technical details are known, but no exploit is available. The price for an exploit might be around USD $0-$5k at the moment (estimation calculated on 06/05/2019).

The vulnerability was handled as a non-public zero-day exploit for at least 2 days. During that time the estimated underground price was around $25k-$100k. The vulnerability scanner Nessus provides a plugin with the ID 19133 (FreeBSD : php -- multiple vulnerabilities (d47e9d19-5016-11d9-9b5f-0050569f0001)), which helps to determine the existence of the flaw in a target environment. It is assigned to the family *FreeBSD Local Security Checks* and running in the context local.

‘Multiple Vulnerabilities within PHP 4/5 (pack, unpack, safe_mode_exec_dir, safe_mode, realpath, unserialize)’

Published on December 16th, 2004

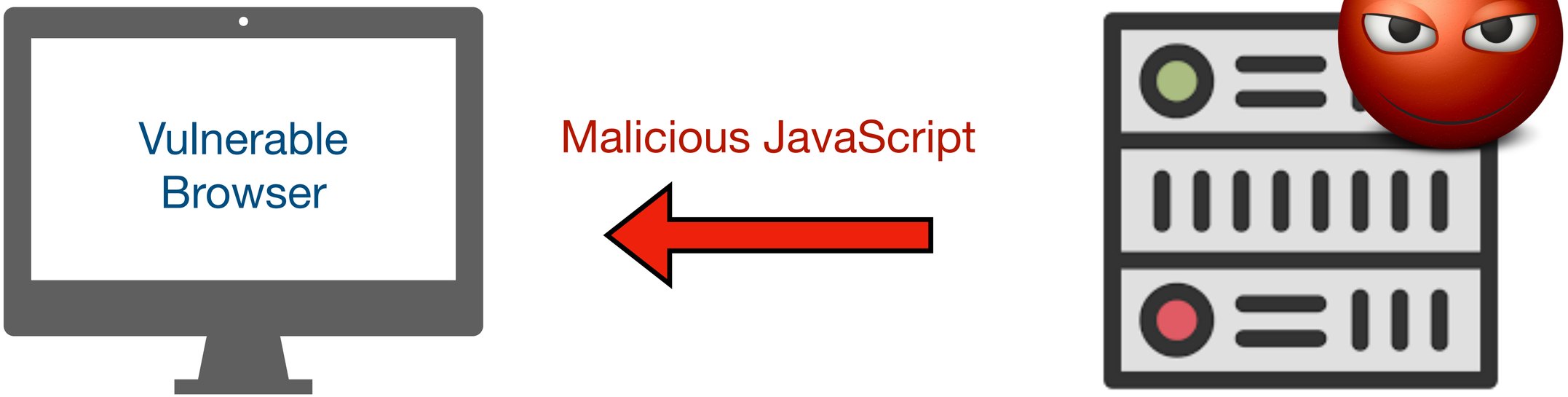
```
[06 - unserialize() - wrong handling of negative references ]

The variable unserializer could be fooled with negative references
to add false zvalues to hashtables. When those hashtables get
destroyed this can lead to efree()s of arbitrary memory addresses
which can result in arbitrary code execution. (Unless Hardened-PHP's
memory manager canaries are activated)

[07 - unserialize() - wrong handling of references to freed data ]

Additionally to bug 07 the previous version of the variable
unserializer allowed setting references to already freed entries in
the variable hash. A skilled attacker can exploit this to create
an universal string that will pass execution to an arbitrary
memory address when it is passed to unserialize(). For AMD64 systems
a string was developed that directly passes execution to code
contained in the string itself.
```

Creating an exploit

- Mastering low-level Assembly programming & debugging of binary code
- Understanding how to trigger the overflow in a controlled setting
- Replicating the execution environment of the target code:
    1. Predicting the address of which we want to gain control
    2. Predicting the address to which the shellcode will be written
    3. Avoiding crashes before taking control
- What if this is not possible? ⇒ Many tricks and workarounds

Browser Memory Model

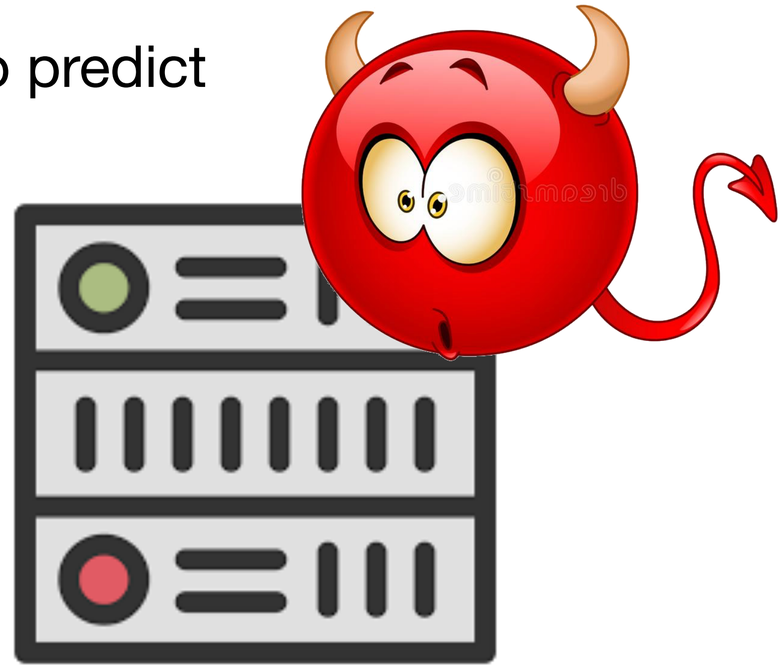
- Malicious web server wants to gain control of client machines
  - E.g., to install malware, etc.
- It can execute JavaScript code in the client machine!



Browser Memory Model

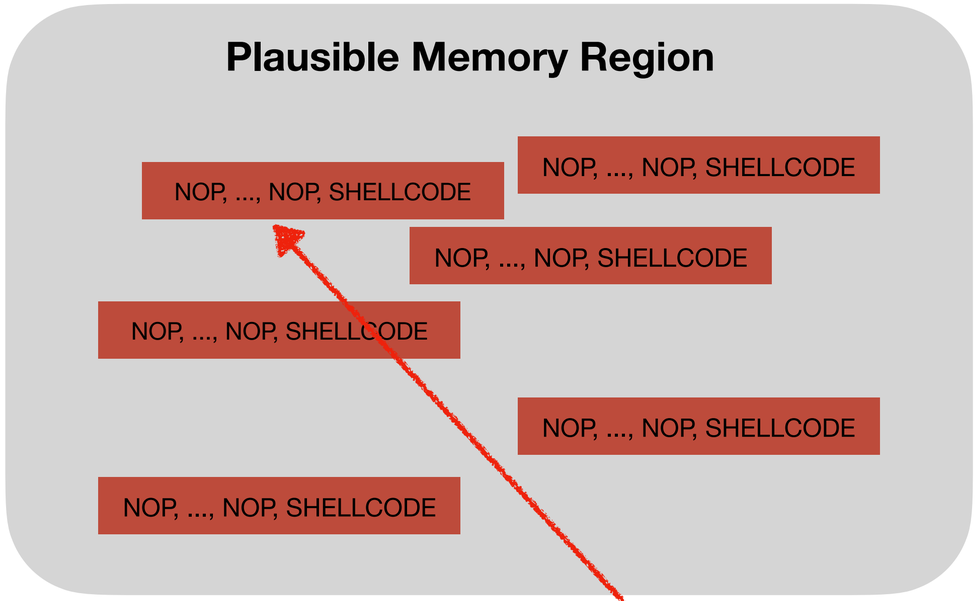
- Challenge:
  - A browser is almost a new kind of “OS”
  - Its behaviour depends on several factors, hard to replicate
  - The layout of injected code in memory is very hard to predict

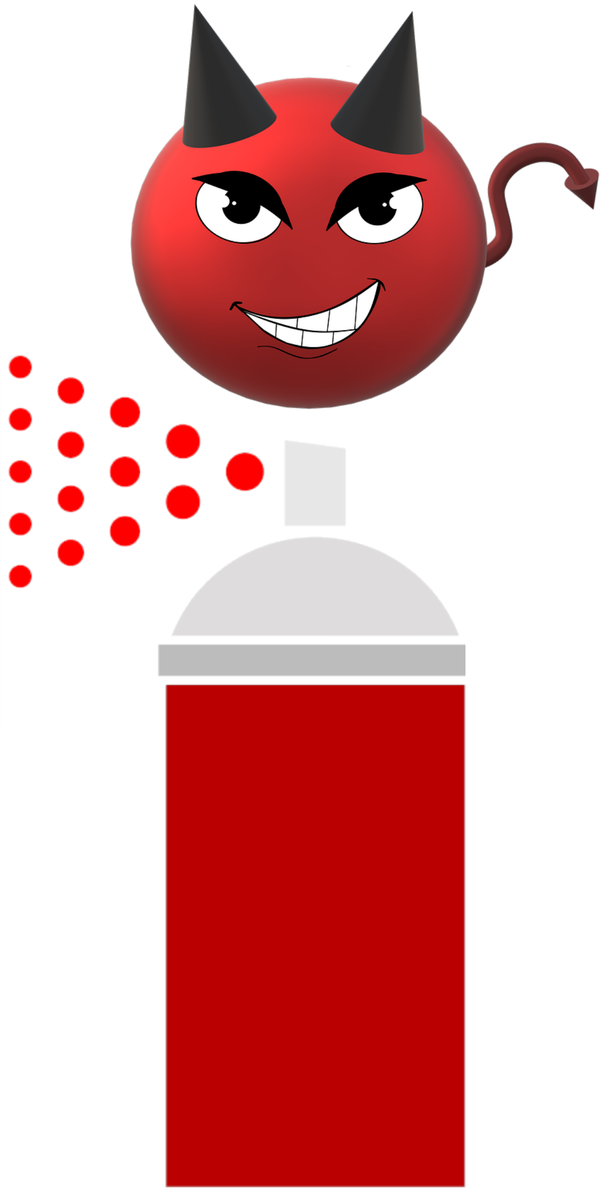
Malicious JavaScript



“Heap Spraying”

- **Adversaries can cheat!**
- Malicious JavaScript floods the memory with copies of shellcode + NOP sled
- Goal: redirect pointers to the “sprayed” memory region
- Increases changes of gaining control





“Heap Spraying” Example

Microsoft Internet Explorer javaprxy.dll COM Object Vulnerability

Title : Microsoft Internet Explorer javaprxy.dll COM Object Vulnerability
Advisory ID : FrSIRT/ADV-2005-0935
CVE ID : CVE-2005-2087
Rated as : **Critical**
Remotely Exploitable : Yes
Locally Exploitable : Yes
Release Date : 2005-07-01

Advisory Details

- Description
- Affected Products
- Solution
- References

Technical Description

A vulnerability was identified in Microsoft Internet Explorer, which could be exploited by remote attackers to execute arbitrary commands. This flaw is due to an error in the "javaprxy.dll" COM Object when instantiated in Internet Explorer via a specially crafted HTML tag, which could be exploited via a malicious Web page to compromise and take complete control of a vulnerable system.

Proof of concept Exploit :
http://www.frsirt.com/exploits/20050702.iejavaprxyexploit.pl.php

Credits

Vulnerability reported by Bernhard Müller and Martin Eiszner

ChangeLog

2005-07-01 : Initial release
2005-07-02 : Exploit available
2005-07-05 : Updated CVE
2005-07-05 : Updated Solution (KB903235 tool)
2005-07-12 : Updated Solution (MS05-037)

Other Kinds of Overflows

Use After Free

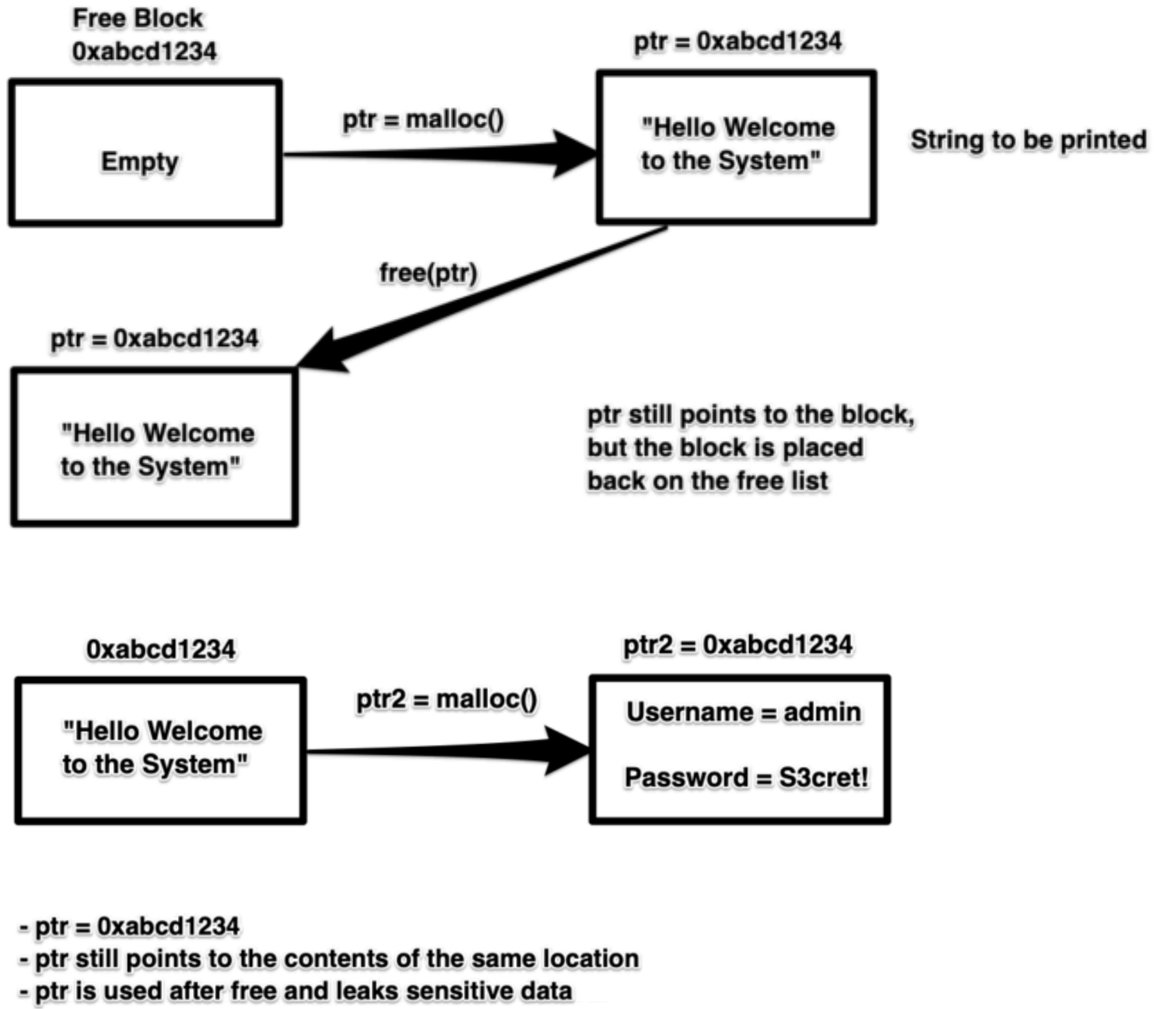
- What if:
  - In languages with dynamic memory management (e.g., C), the code accesses a previously freed memory region?
  - In object-oriented languages (e.g., C++), the code calls a method of an object's destroyed instance?
- May not crash if memory remains stable ⇒ **unpredictable behaviour** 
- If adversary can place his input in such memory ⇒ **shellcode execution** 

Use After Free

- Classical “dangling pointer”

```
char* ptr = malloc(100);
free(ptr);
char* ptr2 = malloc(100);
strcpy(ptr2,bad);
puts(ptr); //dangling pointer
```

- At the end:
  - Where does `ptr` point to?
  - What is stored in `ptr2`?
  - For sure?



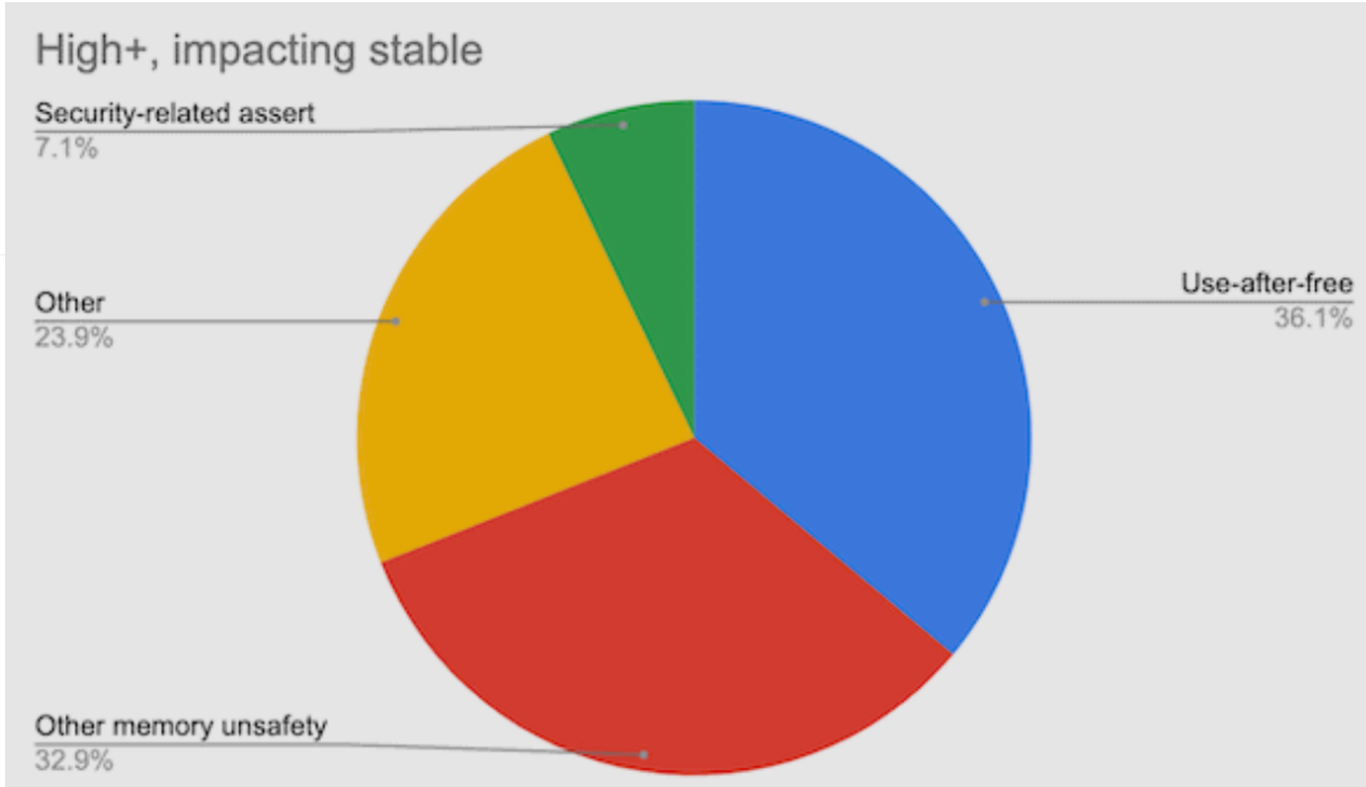
[https://bufferoverflows.net/use-after-free-vulnerability-uaf/]

Memory Errors in Browsers

Chrome: 70% of all security bugs are memory safety issues

Google software engineers are looking into ways of eliminating memory management-related bugs from Chrome.

By Catalin Cimpanu for Zero Day | May 23, 2020 -- 06:00 GMT (07:00 BST) | Topic: Security



Integer Overflow

- When the machine's integer representation causes **loss of information**
  - E.g., using 4 bytes to store a `int` in C

```
int i = 123456789;
```

  - E.g., truncating when casting into a type with a smaller size

```
int i = 0x12345678;
short s = i;
char c = i;
```

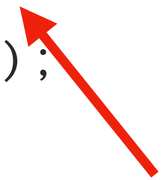
Integer Overflow

- May be subtle: different numerical representations (`unsigned short` << `size_t`)

```
struct s {
      unsigned short len;
      char buf[];
};

void foo(struct s *p) {
       char buffer[100];
       if (p->len < sizeof buffer)
              strcpy(buffer, p->buf);
              // Use buffer
}
```

```
int main(int argc, char *argv[]) {
       size_t len = strlen(argv[0]);
       struct s *p = malloc(len + 3);
       p->len = len;
       strcpy(p->buf, argv[0]);
       foo(p);
       return 0;
}
```

may not fit >>>



may hold even with strlen(p->buf) >> 100

Integer Overflow

- Arithmetic modulo *n* bits
  - Unexpected results due to bounded arithmetic ⇒ overflow

```
unsigned int product = a * b;
unsigned int sum = a + b;
unsigned int difference = a - b;
```

  - E.g., what is the result of `(char) 0xff + 0x1`?
  - If we are computing the size of a new memory region we may allocate 0 bytes instead of 256 bytes!

Integer Overflow

- Even more subtle cases may forget signal representations

```
if (x < 100) // followed by some code that uses x
```

  - What if x is `0xffffffff`? It represents -1 in words of 32 bits!

- Or compare values with different signal representations

  - What if size < 0? We allocate 0 bytes!

```
int size;
if (size < sizeof(x))
  p = malloc(size);
```

Creating an exploit

1. If adversary controls an integer in the input
2. It may cause an integer overflow in an intermediate computation
3. Triggering the allocation of a smaller than expected buffer
4. Leading to a buffer overflow
5. Can potentially execute arbitrary code or cause a crash
   (DoS is a valid attack!)

OpenSSH 3.3 (2002)

```
nresp = packet_get_int();
if (nresp > 0) {
        response = xmalloc(nresp*sizeof(char*));
        for (i = 0; i < nresp; i++)
                response[i] = packet_get_string(NULL);
}
```

- Attacker controls `nresp`
- Input is copied byte-by-byte
- Overflow in the multiplication leads to allocating 0 bytes

ImageMagick Integer/Buffer Overflows in Processing XCF and Sun Bitmap Images Lets Remote Users Execute Arbitrary Code

SecurityTracker Alert ID: 1016749

SecurityTracker URL: http://securitytracker.com/id/1016749

CVE Reference: CVE-2006-3743, CVE-2006-3744 *(Links to External Site)*

Date: Aug 24 2006

Impact: Execution of arbitrary code via network, User access via network

Fix Available: Yes **Vendor Confirmed:** Yes

Version(s): 6.2.9 and prior versions

Description: A vulnerability was reported in ImageMagick. A remote user can cause arbitrary code to be executed on the target user's system.

A remote user can create a specially crafted XCF or Sun bitmap graphic file that, when loaded by the target user, will trigger a buffer overflow and execute arbitrary code on the target system.

Impact: A remote user can create an image file that, when loaded by the target user, will execute arbitrary code on the target user's system.

Solution: The vendor has issued a fixed version (6.2.9-1), available at:

Out-of-bounds Read

- Lower limit (e.g., negative values)

```
int getValueFromArray(int *array, int len, int
index) {
  int value;
  if (index < len) {
    value = array[index];
  }
  else {
    printf("Value is: %d\n", array[index]);
    value = -1;
  }
  return value;
}
```

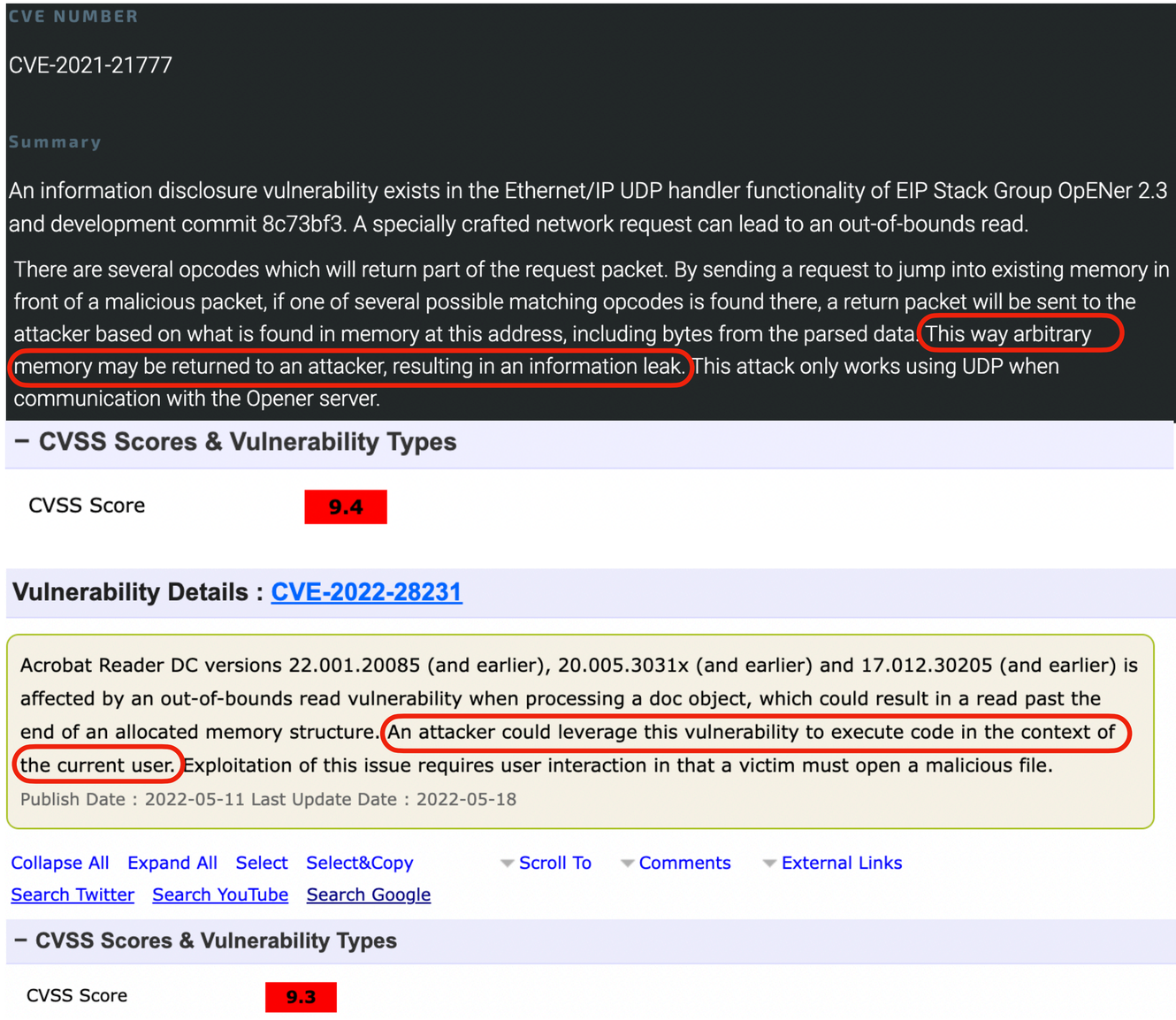
- Upper limit (e.g., off-by-one)

```
int main(void)
{
    int array[5] = {0, 1, 2, 3, 4};
    int i;

    for (i = 0; i <= 5; i++)
        printf("array[%d] = %d\n", i, array[i]);

    printf("array[%d] = %d\n", i, array[i]);

    return 0;
}
```

- Real vulnerabilities much more complex
- **Attacks do not require writing into memory!**

Out-of-bounds Read

- *Out-of-bounds read* vulnerabilities are sufficient for real attacks
- Potential threats:
  - Reading sensitive data
  - Executing code



CVE NUMBER

CVE-2021-21777

Summary

An information disclosure vulnerability exists in the Ethernet/IP UDP handler functionality of EIP Stack Group OpENer 2.3 and development commit 8c73bf3. A specially crafted network request can lead to an out-of-bounds read.

There are several opcodes which will return part of the request packet. By sending a request to jump into existing memory in front of a malicious packet, if one of several possible matching opcodes is found there, a return packet will be sent to the attacker based on what is found in memory at this address, including bytes from the parsed data. This way arbitrary memory may be returned to an attacker, resulting in an information leak. This attack only works using UDP when communication with the Opener server.

– CVSS Scores & Vulnerability Types

CVSS Score 9.4

Vulnerability Details : CVE-2022-28231

Acrobat Reader DC versions 22.001.20085 (and earlier), 20.005.3031x (and earlier) and 17.012.30205 (and earlier) is affected by an out-of-bounds read vulnerability when processing a doc object, which could result in a read past the end of an allocated memory structure. An attacker could leverage this vulnerability to execute code in the context of the current user. Exploitation of this issue requires user interaction in that a victim must open a malicious file.

Publish Date : 2022-05-11 Last Update Date : 2022-05-18

Collapse All Expand All Select Select&Copy Scroll To Comments External Links

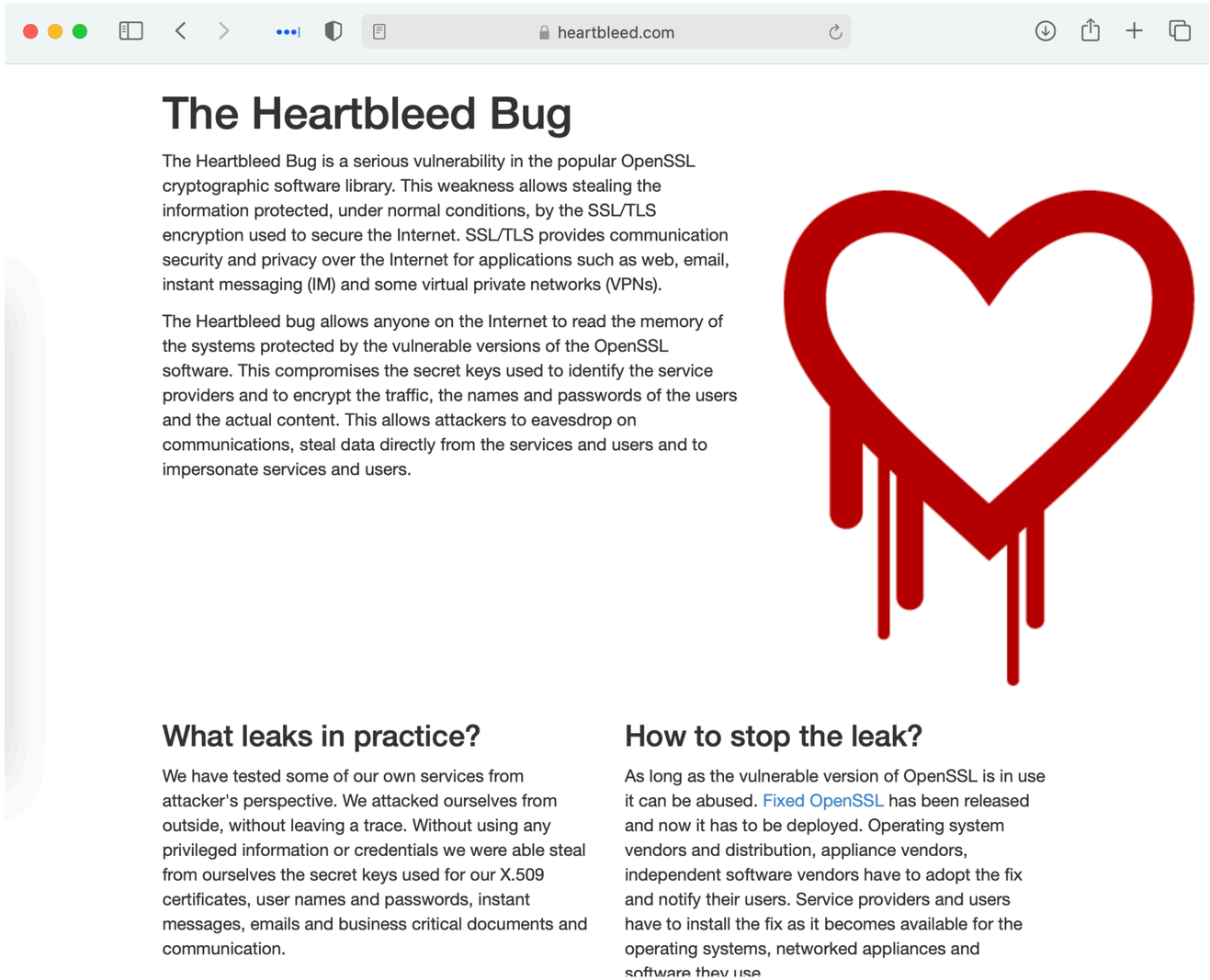
Search Twitter Search YouTube Search Google

– CVSS Scores & Vulnerability Types

CVSS Score 9.3

HeartBleed (2014)

- “heartbeat” TLS: client sends a small message to the server to keep the session alive
- Message contains an integer (fixed in the standard) that describes the length of the transmitted data
- OpenSSL implementation **reuses the value given by the client** to count how many bytes are read from memory
- Result: **malicious client can obtain a response with sensitive information** that resides in the server’s memory (e.g., passwords, crypto keys, etc)



Format Strings

- Consider the ubiquitous `printf` function
- Receives as first argument a ***format string***:

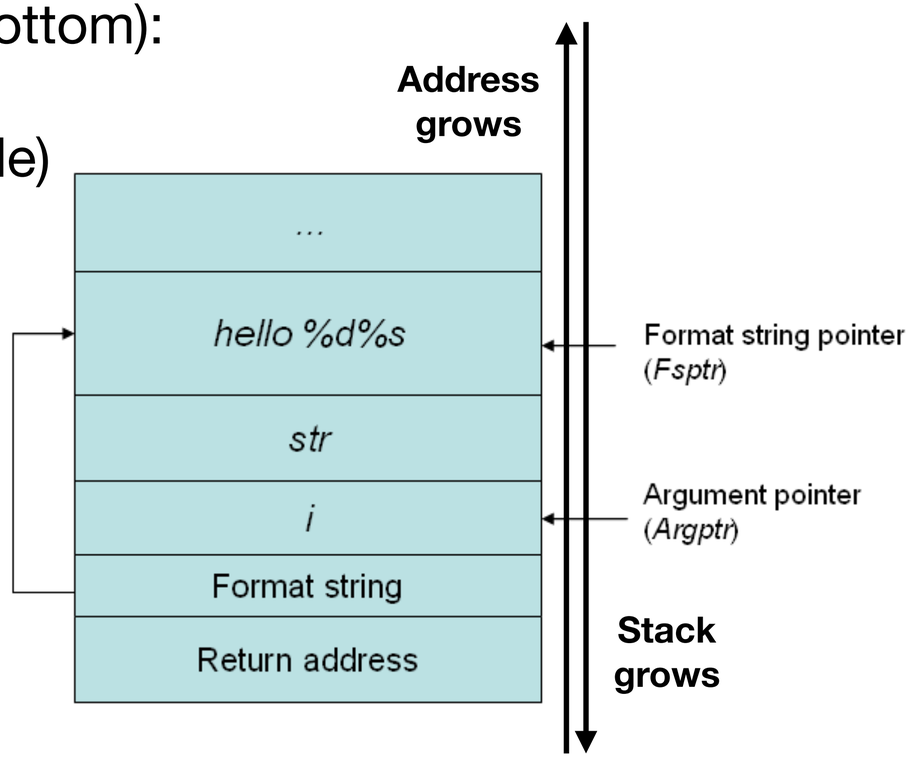
```
printf(“An integer %d followed by a string %s\n", i, s)
```

- **And how many more parameters?**
  - It depends …
  - How is it implemented? C supports variadic functions (`va_list`, `va_arg`, `va_start`, `va_copy`, `va_end`)
  - Basically, **the parameters go into the stack** ⇒ 

Format Strings

```
printf("An integer %d followed by a string %s\n", i, s)
```

- The parameters are placed in the stack (drawn from top to bottom):
  - format string: array of characters in the stack (local variable)
  - integer `i` and string `s` (address) in reverse order
  - address of the format string
- `printf` traverses the format string and
  reads arguments in immediately higher stack addresses!
- What happens if there are less/more arguments than %?



https://www.researchgate.net/figure/Stack-of-the-printf-function-call_fig2_220885359

Format Strings

- Imagine a classical mistake:

```
printf("%s\n", s)
printf(s) // forgot the format string
```

- What is going to happen?
- If the attacker controls the format string:
  - `printf("%d\n")` prints something directly from the stack
  - `printf("%s\n")` prints something pointed to by an address in the stack
  - `printf("<address>%d%d%d...%s")` allows reading any `<address>` from memory! $\Longrightarrow$ **The format string itself is in the stack!!!**

Format Strings

- Can we only read from arbitrary memory?
- No: `if (strlen(src) < sizeof(dst)) sprintf(dst, src);`
- `sprintf` writes to a memory array
- If `src` is controlled by the adversary and contains “`%s`”
  - What is the size of the written region? $\Longrightarrow$ the size of the source string `src`!
  - Note that the adversary may simply control the format string parameters, not `src`

Format Strings

`n`	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.

- An even more radical option exists: `%n`
- The idea is to store in the argument address the number of printed characters so far
- `printf` is Turing-complete! https://github.com/carlini/printf-tac-toe
- This option is excluded from most platforms:
  - If the adversary controls the format string he can write to arbitrary memory
  - What if he wants to write to address `0xFFFFFF`? 2^{32} characters? There are tricks: `%*`

Acknowledgements

- This lecture's slides have been inspired by the following lectures:
  - CSE127: Low-level Software Security III: Integer overflow, ROP and CFI
  - CS155: Basic Control Hijacking Attacks
  - CS343: Integer overflows + Format string vulnerabilities + Heap control data