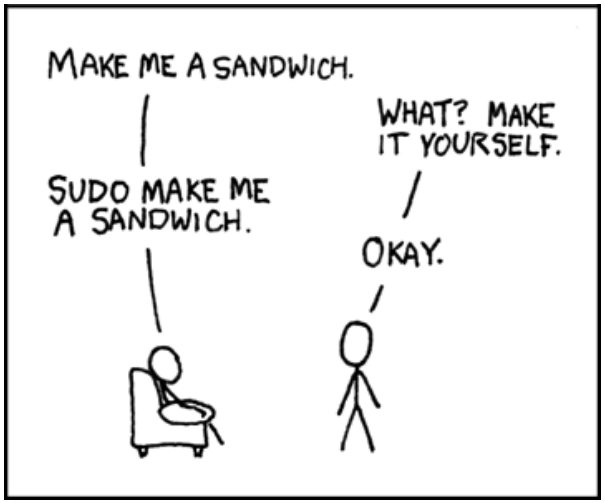
Sandwiches for everyone

Stratis Viglas

Inf2C :: Computer Systems



Today's menu ("And finally, monsieur, a wafer-thin mint")

- Notes on *security*
  - Or, why *safety* is an *illusion*, why *ignorance* is *bliss*, and why *knowledge* is *power*
- *Stack* overflows
  - Or, why you were *doing it wrong* in the first assignment
- *Null* pointers
  - Or, why everyone can be *stupid*
  - Kinda like calling a FIFO data structure a stack[1]
- *Demos!*
  - Or, another reason why this is likely my *last Inf2C lecture*

[1] Your lecturer, 17/11/09, 3:38pm

Today's menu ("And finally, monsieur, a wafer-thin mint")

- Notes on *security*
  - Or, why *safety* is an *illusion*, why *ignorance* is *bliss*, and why *knowledge* is *power*
- *Stack* overflows
  - Or, why you were *doing it wrong* in the first assignment
- *Null* pointers
  - Or, why everyone can be *stupid*
  - Kinda like calling a FIFO data structure a stack[1]
- *Demos!*
  - Or, another reason why this is likely my *last Inf2C lecture*

Please, pretty please, with sugar on top

Try it at home, not on DICE (well, not too frequently anyway)

[1] Your lecturer, 17/11/09, 3:38pm

CPU/OS (Linux) security model

- The *rings* of fire[2] (offered by the CPU)
  - *Four rings*, zero to three
- As usual, one ring *rules them all*[3]
  - *ring0* is *kernel* space, *ring3* is *user* space
  - *ring1* and *ring2* are *not used* by the OS
- We need to be in *ring0* to be `root`
- We'll *play around* with *memory locations* and we'll *dereference* some *null pointers* in the process to get into *ring0* from *ring3*

[2] Johnny Cash $\gg$ J. R. R. Tolkien

[3] By the way, I hate Tolkien

The call/execution stack

- *You've used it* in MIPS

The call/execution stack

- *You've used it* in MIPS
  - And *you abused* it
  - *You were hacking* it without even knowing
  - This made me happy in a weird way
- It all comes down to *four calls* in x86 assembly
  - `push src`: push value in `src` onto the stack
  - `pop dst`: pop from stack and store in `dst`
  - `call loc`: *call* a *function* stored in `loc` and *push* the *return address* onto the stack (*i.e.*, the next value of the program counter)
  - `ret`: *return* from a function by *popping* the return address from stack and *jumping* to it

The call/execution stack

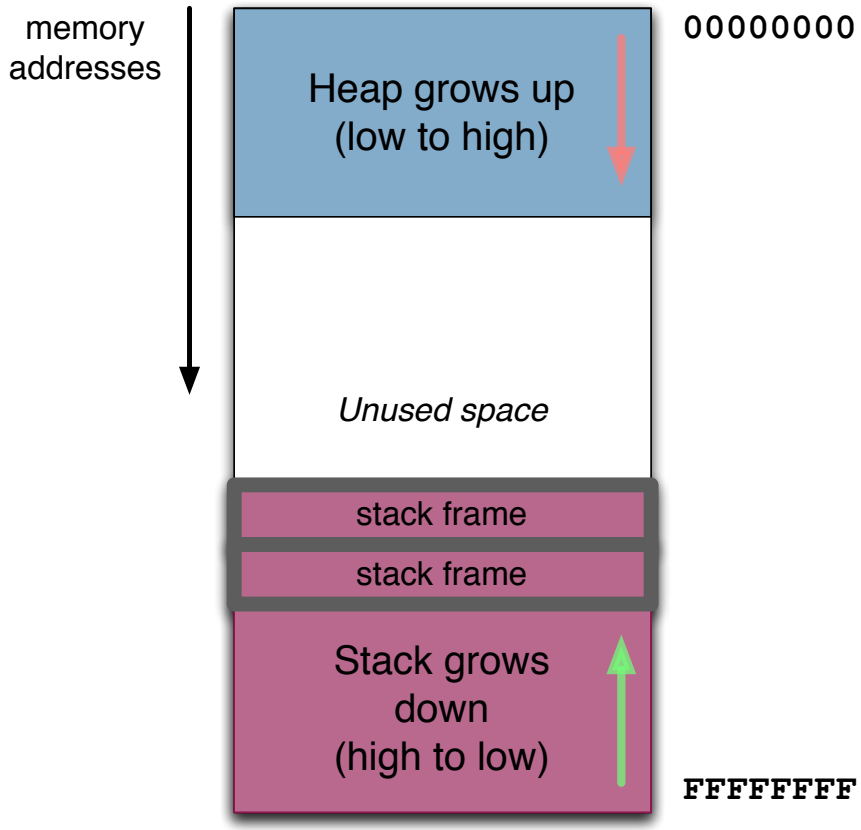
- *You've used it* in MIPS
  - And *you abused* it
  - *You were hacking* it without even knowing
  - This made me happy in a weird way
- It all comes down to *four calls* in x86 assembly
  - `push src`: push value in `src` onto the stack
  - `pop dst`: pop from stack and store in `dst`
  - `call loc`: *call* a *function* stored in `loc` and *push* the *return address* onto the stack (*i.e.*, the next value of the program counter)
  - `ret`: *return* from a function by *popping* the return address from stack and *jumping* to it

The ghost in the machine

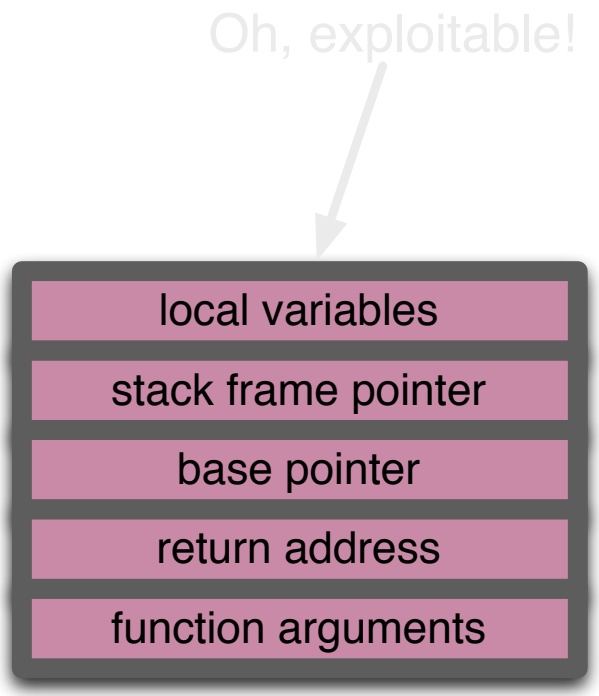
By overwriting the stored return address on the top of the stack before the `ret` call is issued, we take control of program execution

x86 memory management

- Before exploiting memory (mis)management we need to *know how* the *CPU* and the *OS* view memory
- Each *combination* of *architecture* and *operating system* is *different*
  - Though they can all *fail* in similar ways

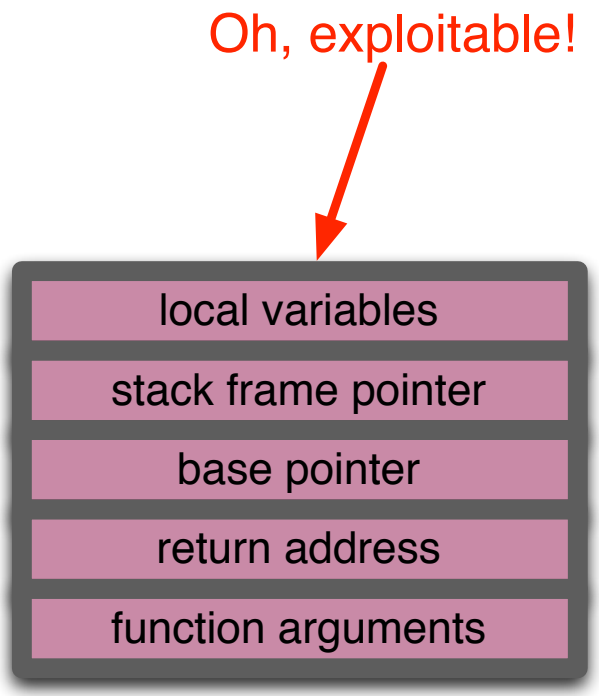


Anatomy of a stack frame



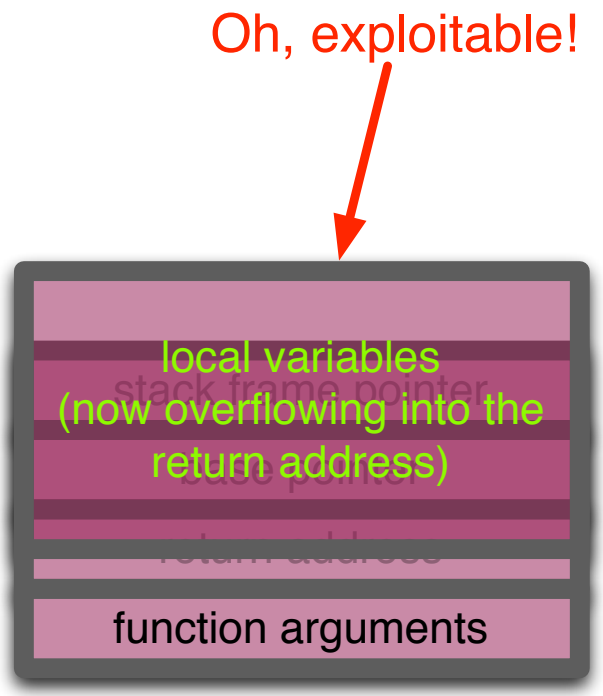
- The *order* of allocations gives way to *overflow and exploits*
  - Function arguments
  - Address to return to
  - Base offset of the program
  - Base of frame

Anatomy of a stack frame



- The *order* of allocations gives way to *overflow and exploits*
  - Function arguments
  - Address to return to
  - Base offset of the program
  - Base of frame
  - *Local function variables*

Anatomy of a stack frame



- The *order* of allocations gives way to *overflow and exploits*
  - Function arguments
  - Address to return to
  - Base offset of the program
  - Base of frame
  - *Local function variables*

A recipe for stack overflows and code embedding

1. *C code* that will *cause* the *overflow* (a redirected return from a function call, or an out-of-bounds `strcpy()` usually do the trick)
2. *Assembly code* that will *execute* once you have *overflowed* the stack
3. Pass the *assembly code* through the *assembler*
4. *Link it* to pick up any stray *library calls*
5. *Dump* the *binary* to *text*
6. *Copy* the *text* of the binary in the *C source file* and *compile* it *disabling* the compiler's *stack protection*
7. *Run* it
8. You now *control* the execution *flow*

A recipe for stack overflows and code embedding

1. *C code* that will *cause* the *overflow* (a redirected return from a function call, or an out-of-bounds `strcpy()` usually do the trick)
2. *Assembly code* that will *execute* once you have *overflowed* the stack
3. Pass the *assembly code* through the *assembler*
4. *Link it* to pick up any stray *library calls*
5. *Dump* the *binary* to *text*
6. *Copy* the *text* of the binary in the *C source file* and *compile* it *disabling* the compiler's *stack protection*
7. *Run* it
8. You now *control* the execution *flow*

Hacking does not mean the names are more imaginative . . .

The process is usually referred to as *shellcoding* or *stack smashing*

The smallest stack overflowing C program (shellcode.c)

```
char code[] = "Your␣shellcode␣here";

int main(int argc, char **argv)
{
  int (*func)();                  // declare a function
                                  // pointer returning an int
  func = (int (*)()) code;        // turn the string into
                                  // a function
  (int)(*func)();                 // call it to smash the stack
                                  // why? because the place we
                                  // jump to is a string
                                  // which we will execute
}
```

Subliminal message

C/C++ are really the only languages

Hello world in assembly (hello.asm)

```
;hello.asm
[SECTION .text]
global _start
_start:     jmp short ender
starter:    xor eax, eax     ;clean up the registers
            xor ebx, ebx
            xor edx, edx
            xor ecx, ecx
            mov al, 4        ;syscall write
            mov bl, 1        ;stdout is 1
            pop ecx          ;address of string from stack
            mov dl, 5        ;length of the string
            int 0x80
            xor eax, eax
            mov al, 1        ;exit the shellcode
            xor ebx,ebx
            int 0x80
ender:      call starter     ;address of string on stack
            db 'hello'
```

Object dump

```
hello:     file format elf32-i386
Disassembly of section .text:
08048060 <_start>:
 8048060: eb 19                 jmp    804807b <ender>
08048062 <starter>:
 8048062: 31 c0                 xor    %eax,%eax
 8048064: 31 db                 xor    %ebx,%ebx
 8048066: 31 d2                 xor    %edx,%edx
 8048068: 31 c9                 xor    %ecx,%ecx
 804806a: b0 04                 mov    0x4,%al
 804806c: b3 01                 mov    0x1,%bl
 804806e: 59                    pop    %ecx
 804806f: b2 05                 mov    0x5,%dl
 8048071: cd 80                 int    0x80
 8048073: 31 c0                 xor    %eax,%eax
 8048075: b0 01                 mov    0x1,%al
 8048077: 31 db                 xor    %ebx,%ebx
 8048079: cd 80                 int    0x80
0804807b <ender>:
 804807b: e8 e2 ff ff ff        call   8048062 <starter>
 8048080: 68 65 6c 6c 6f        push   0x6f6c6c65
```

The code

See the funny little hexadecimals? That's the code that will be executed once the stack has overflowed. Now, let's turn this into a string ...

(Not so) Beautiful code

```
char code [] = "\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31"\
               "\xc9\xb0\x04\xb3\x01\x59\xb2\x05\xcd"\
               "\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80"\
               "\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f";

int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```

Putting it all together

```
sviglas@munch:~/exploits$ emacs shellcode.c
<clicky clicky clicky>
sviglas@munch:~/exploits$ emacs hello.asm
<clicky clicky clicky>
sviglas@munch:~/exploits$ nasm hello.asm
sviglas@munch:~/exploits$ ld -o hello hello.o
sviglas@munch:~/exploits$ objdump -d hello
<a whole bunch of hexadecimals>
sviglas@munch:~/exploits$ emacs shellcode.c
<clicky clicky clicky>
sviglas@munch:~/exploits$ gcc -fno-stack-protector \
     -o shellcode shellcode.c
sviglas@munch:~/exploits$ ./shellcode
hello
```

Subliminal message

Use emacs; get finger cramps like all the cool kids do

The possibilities are endless

- *Every piece of code* you execute *uses* the *stack*
  - So *every piece of code* you write can *cause* a stack *overflow*
  - You should **really** stop complaining about the marks of the first assignment!

The possibilities are endless

- *Every piece of code* you execute *uses* the *stack*
  - So *every piece of code* you write can *cause* a stack *overflow*
  - You should **really** stop complaining about the marks of the first assignment!
- What if the *code* you execute creates a *copy* of *itself*?
- *Remotely*, from a socket
  - *Internet worms* did exactly that: they *smashed* the stack and *reproduced*

The possibilities are endless

- *Every piece of code* you execute *uses* the *stack*
  - So *every piece of code* you write can *cause* a stack *overflow*
  - You should **really** stop complaining about the marks of the first assignment!
- What if the *code* you execute creates a *copy* of *itself*?
- *Remotely*, from a socket
  - *Internet worms* did exactly that: they *smashed* the stack and *reproduced*
- What if it *spawns* some *shell* that should be run by `root`?
  - *Piece of cake* if the stack smashing *process* is *owned by root* (*e.g.*, overflowing due to a ridiculously long command line argument to, say, `passwd`)
  - Check the `setreuid()` documentation
  - *Not so easy* in user space, but not insanely hard either
- An idea: *system call* to request from an *application* that is run by `root` (servers are like that) to *dynamically load* a piece of *code* that *smashes* the stack and in doing so *spawns a shell*

Straight from the kernel

```
static unsigned int tun_chr_poll(struct file *file,
                                 poll_table * wait)
{
  struct tun_file *tfile = file->private_data;
  struct tun_struct *tun = __tun_get(tfile);
  struct sock *sk = tun->sk;    // assignment of the pointer
                                // before test for NULL
  unsigned int mask = 0;

  if (!tun) return POLLERR;     // pointer has already been
                                // dereferenced; -O3 in gcc
                                // will take this test out

  // make tun->sk point to 0x00000000, a valid address
  // sk is now under our control, owned by root,
  // and in kernel space
```

The recipe

1. *Rely* on the *compiler* to "optimise" the code and *take out* certain `NULL` pointer *checks*
   - *Assignment* of a *pointer* to *another pointer* if the *first pointer* is already known to be *not*-`NULL`, makes a *check* for `NULL`-*ness* of the second pointer *obsolete*
2. *Nullify* a *pointer* by making it *point* to *address zero* (this is not a null pointer, this is a perfectly *valid assignment*)
3. Create an *OS page* and populate it with the *exploit* from user space
   - Most *likely* the exploit is `/bin/sh`
   - *Page* needs to be *owned by root*, so *use* a *root-run service* to *dynamically load* it (*e.g.*, `pulseaudio`)
4. *Map* the page to the pointer's *address*
   - The *compiler will not check*; it's a *valid pointer*
   - The *kernel will not complain*; it *"owns" the page*
5. *Assign* that value to a *kernel-controlled* pointer in *ring0*

The recipe

1. *Rely* on the *compiler* to "optimise" the code and *take out* certain `NULL` pointer *checks*
   - *Assignment* of a *pointer* to *another pointer* if the *first pointer* is already known to be *not*-`NULL`, makes a *check* for `NULL`-*ness* of the second pointer *obsolete*
2. *Nullify* a *pointer* by making it *point* to *address zero* (this is not a null pointer, this is a perfectly *valid assignment*)
3. Create an *OS page* and populate it with the *exploit* from user space
   - Most *likely* the exploit is `/bin/sh`
   - *Page* needs to be *owned by root*, so *use* a *root-run service* to *dynamically load* it (*e.g.*, `pulseaudio`)
4. *Map* the page to the pointer's *address*
   - The *compiler will not check*; it's a *valid pointer*
   - The *kernel will not complain*; it *"owns" the page*
5. *Assign* that value to a *kernel-controlled* pointer in *ring0*

Guess what?

You are `root`

Further reading

- Smashing the stack for fun and profit [Aleph One, Phrack, 1996]
- Hacking: the Art of Exploitation [Erickson, No Starch, 2008]
- The linux kernel [Torvalds *et al.*, ongoing]
- You can do the same things in Windows
  - Though you don't have access to the source code so it needs quite a bit of reverse engineering

So why do this?

So why do this?

- *Not* for the *lulz*[4], or for *great justice*[5]

[4] `http://encyclopediadramatica.com`

[5] All your base are belong to us

So why do this?

- *Not* for the *lulz*[4], or for *great justice*[5]
- Do it for the *art*
- Do it because *you can*
- Do it to *learn*

[4] `http://encyclopediadramatica.com`

[5] All your base are belong to us

So why do this?

- *Not* for the *lulz*[4], or for *great justice*[5]
- Do it for the *art*
- Do it because *you can*
- Do it to *learn*
- *Don't* do *harm*

[4] `http://encyclopediadramatica.com`

[5] All your base are belong to us

So why do this?

- *Not* for the *lulz*[4], or for *great justice*[5]
- Do it for the *art*
- Do it because *you can*
- Do it to *learn*
- *Don't* do *harm*

Bottom line

- Bugs and "features": they're everywhere
- Don't be afraid to try

[4] `http://encyclopediadramatica.com`

[5] All your base are belong to us