Shellcode + Exploitation Introduction

pop pop ret

Exploitation!

- Poppin’ a shell, rewtin a box, haqin the gibson.
- We’ll learn how to write basic exploits for low level vulnerabilities,
- For practice, you are going to want to turn off ASLR on linux (as root),

sysctl kernel.randomize_va_space=0

(warning makes your box more hackable..)

- You also want to disable stack cookies when compiling your test things:

-fno-stack-protector at GCC time

Exploitation!

- We’ll do several types of exploitation

- Stack overflow (this week)

- Format string (next week – john)

- Heap overflow (week after – evgeny)

Stack overflows

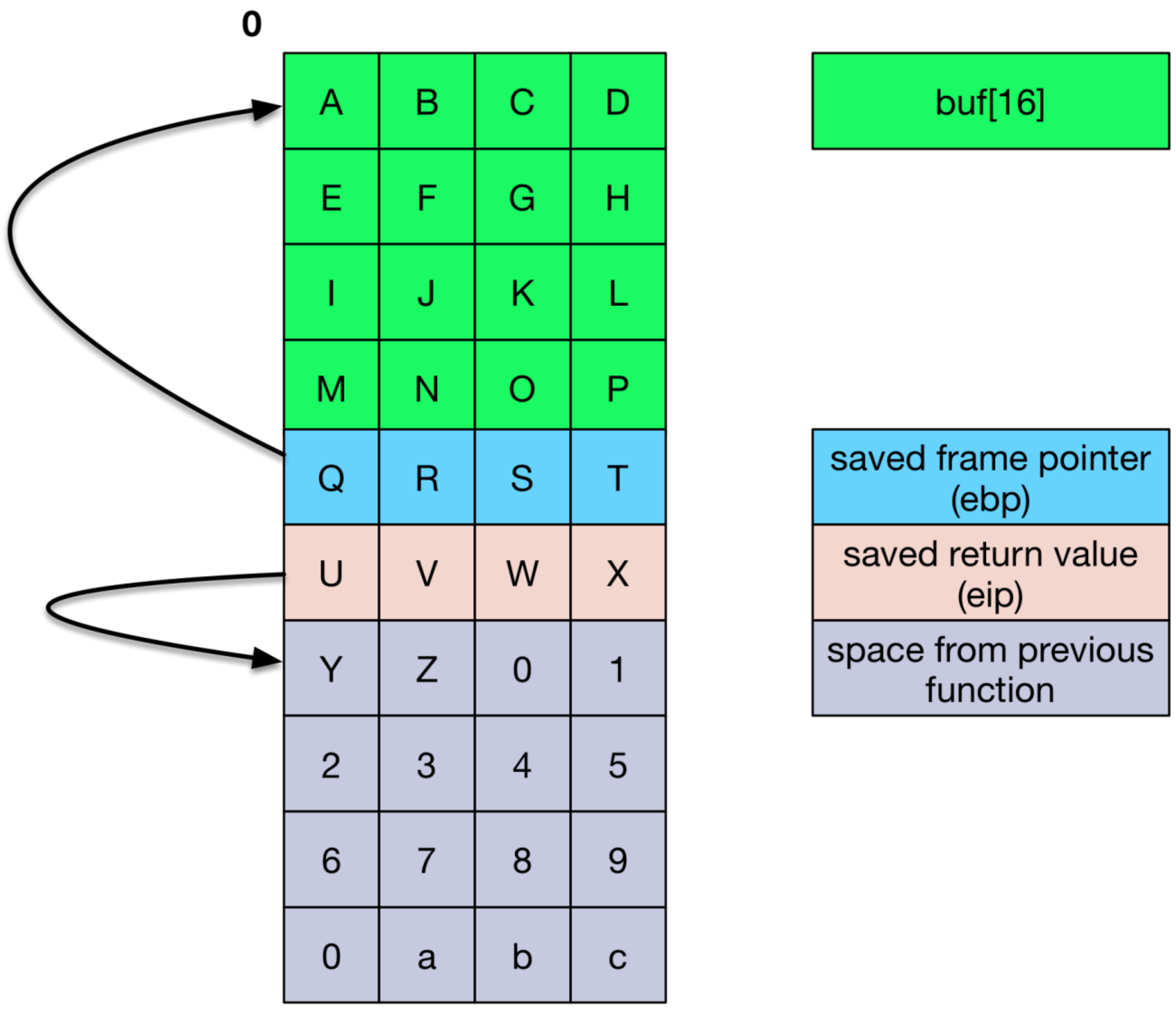
Smashing the stack for fun and profit (1996): http://phrack.org/issues/49/14.html

Our example code:

```
void stupid(char *s) {
    char buf[16];
    strcpy(buf, s);
}
```

Called in next diagram with s being ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abc

Basic Stack Overflow



What is shellcode?

- The payload of your exploit, writing in asm
  - Popping shellz
  - Popping calcz
- Originally called egg code (some guy with eggplant as a nick coined it) but no one knows this
- Now seen as popping a shell

```
char *shellcode =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x6
2\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\x
cd\x80";
```

```
xor eax,eax
push eax
push 0x68732f2f
push 0x6e69622f    // /bin/sh in ascii
mov esp,ebx
push eax
push ebx
mov ecx,esp
mov al,0xb
int 0x80         // execve /bin/sh
```

“How do I get some of those magic bytes?” you ask the Sorcerer

- Write the ASM by hand, assemble using nasm, grab the bytes
  - PITA for complex shellcode
- Write it in C, compile (probably with –static, likely with -Os) and then extract
  -
    ```
    objdump -d ./PROGRAM|grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g’
    ```

syscalls

- We need to interact with the operating system (e.g. open files/exec other programs)
  - This is done with syscalls
- Syscalls are numbered
  - sys_exit == 1 (on x86), sys_fork == 2, etc
- This number is put in eax and then triggered either by an interrupt (int 0x80) or through ‘syscall/sysenter’

sys_exit example

sys_exit takes 1 argument (the exit status code) in ebx

```
xor ebx,ebx
mov eax,0x1
int $0x80
```

```
exit(0);
```

Position Independence

- Your shellcode won't originally know where it is in memory
- Can't hardcode memory addresses
  - Everything has to be relative

```
call stuff
stuff:
pop eax  // eax now has eip
```

NOPpin

- Say you have 20 bytes of shellcode – when you specify an address, you need to land exactly at the start to execute the shellcode. – there is no margin for error.
- What if the program lets you copy in 10 megs? Seems silly that you still have to land it right on the nose.
- 0x90 – NOP – does nothing
- NOPNOPNOPNOPNOP*10million+20 bytes of shellcode = win

NOPpin

- IDS and things like that sometimes actually (lol) alert when they see lots of 0x90 – these can really be any junk 1 byte instructions (or multibyte instructions with no permutation of bad)
- The lots of NOPS thing is called a sled. NOP SLED.

Encoding

- You will often want to encode your shellcode, as you cannot have some characters in your attack
- i.e. no nul bytes in strcpy
- 0xA in a HTTP request ;(
- Common encoders are available in frameworks like metasploit, read the encoders to work out what people are doing
- XOR is very common, for example
- It is good to write your own shellcode, at least once, to work out what is going on
- Most people just maintain collections (or use metasploit)

Common Payloads

- Bindshell
  - Binds to a local port, drops us to a shell when we connect
  - Not great in 2015 due to NAT/firewalls
- Connect back
  - Similar deal, but it connects to us instead
  - Better, but can still be blocked if there is egress filtering (pretty common these days)
  - Can be fancier and connect back via protocols more likely to get back e.g. ICMP/DNS/HTTP(S)

cont

- Socket reuse
  - Dup dup dup
  - We use the same socket we connected to the server with to drop to a shell
  - This should be your server-side goto
- Download and execute
  - Connect to a website/ftp/whatever, grabs binary and ./run
  - Fairly popular for browser exploits
- Process modification / direct payload
- A second exploit and a rootkit..

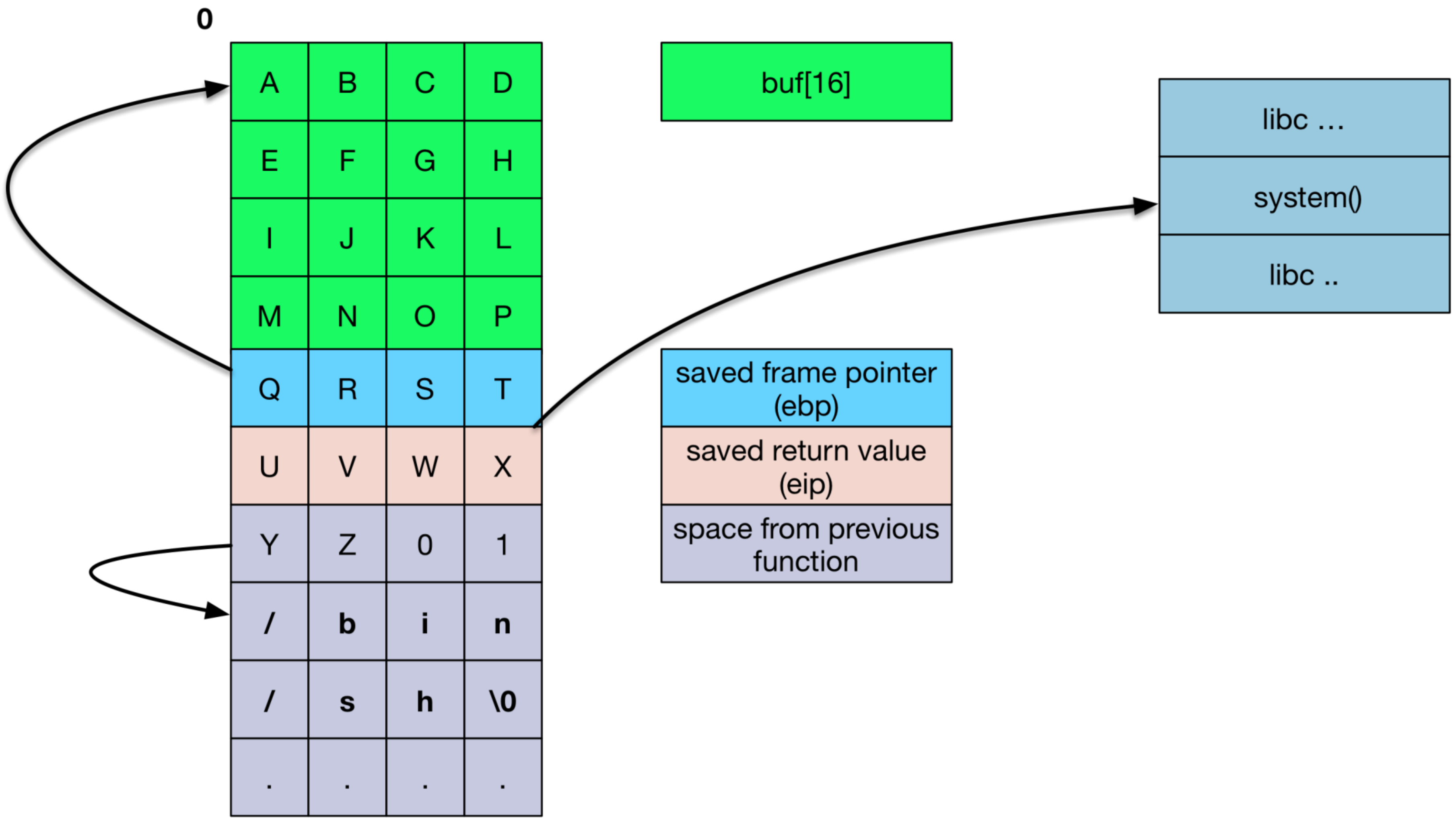
NON EXECUTABLE MEMORY

- Stacks are frequently marked as non-executable (As are loads of other regions)
- Some programs (like javascript engines) still often require executable stacks for performance, though, and things like java definitely do and will for the near future
- Anyway – what can we do if cannot execute shellcode?

RET2LIBC

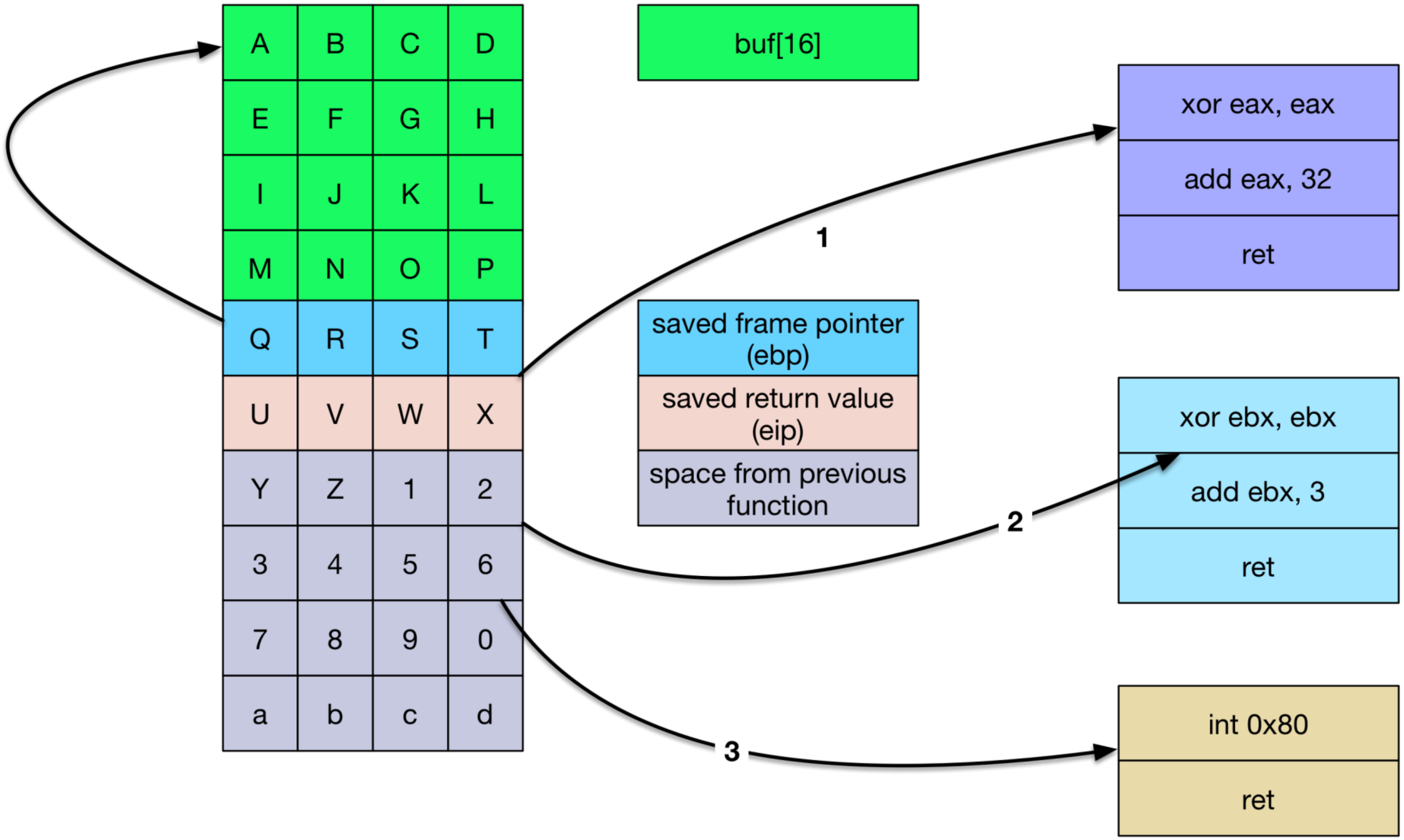
- Solar Designer “Getting around non-executable stack (and fix)” – 1997
- Nergal (The advanced return into lib© exploitation) – 2001 http://phrack.org/issues/58/4.html

RET2LIBC



0
A B C D
E F G H
I J K L
M N O P
Q R S T
U V W X
Y Z 0 1
/ b i n
/ s h \0
. . . .
buf[16]
saved frame pointer (ebp)
saved return value (eip)
space from previous function
libc …
system()
libc ..

ROP



A B C D
E F G H
I J K L
M N O P
Q R S T
U V W X
Y Z 1 2
3 4 5 6
7 8 9 0
a b c d
buf[16]
saved frame pointer (ebp)
saved return value (eip)
space from previous function
1
2
3
xor eax, eax
add eax, 32
ret
xor ebx, ebx
add ebx, 3
ret
int 0x80
ret

ROP || Borrowed Code Chunks

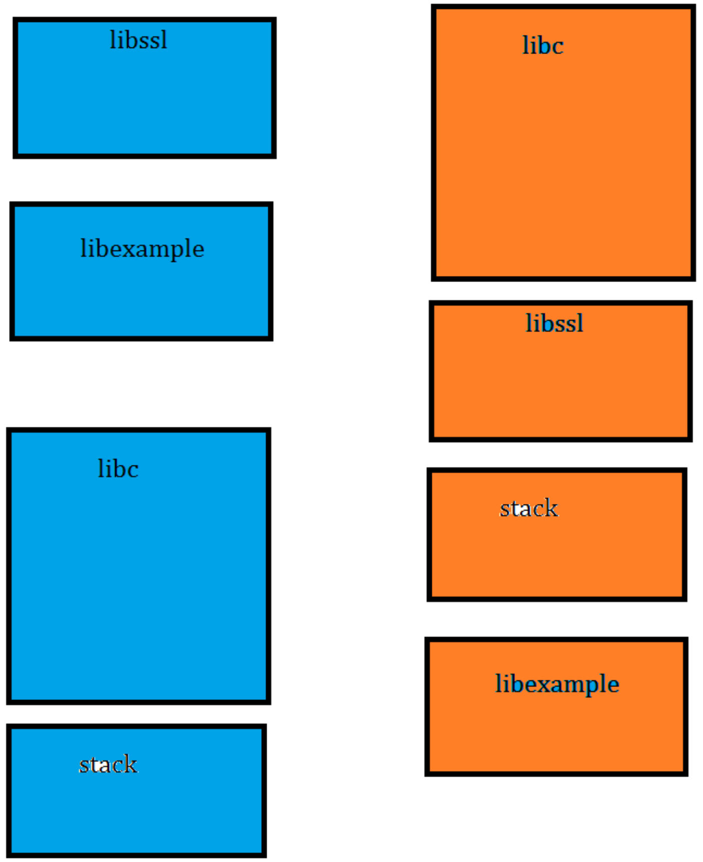
- Return orientated programming or borrowed code chunks means that we are basically creating a fake stack and continuously RETting to gadgets, or pre-existing code chunks, which end with RET or a similar construct
- Doesn’t have to use ret, obviously..
- Made really powerful by massive libraries (even in tiny utilities like our little example, and also various CPU features of intel:
  - Critical instructions being 1 byte, common.
  - So many *!$@!$!8ing instructions
  - Instructions not needed to be aligned..

  - Counter against this is ASLR→ hard to system(“sh”); if you don’t know the address of system..

I don’t quite understand where ret2 ends and ROP starts. I was using assembly snippets as soon as I learned about ret2 out of necessity (we’ll see necessity here, too!)

WHAT ASLR DOES – two execs of the same program

- ASLR means that when SECTIONS become SEGMENTS (file on disk to memory), the base addresses of the segments are randomized.
- That is, things like .text, the stack, the heap, and the base addresses of all the various libraries are at random (page aligned) addresses. They may change order etc.
- We are going to pretend to have full ASLR.



ASLR doesn't:

- Randomize addresses _INSIDE_ libraries. For example, the function "dup2", which we will later use, is at a fixed offset from the base of the library.
- Likewise, variable order in stack/heap is not randomized.
- Maybe in the future every functions address will be randomized; for now., this is too slow and hard..

LIBC

system at base + 1000

dup2 at base + 10404

another_example at base + example score

LIBC

system at base + 1000

dup2 at base + 10404

another_example at base + example score

even though the bases change, the order and relative offset within the sections does not

- What this means:

- If we can leak a pointer to something on the stack, and we know how far that is from something we want to know the address of. Presto.
- Likewise, heap, .text, a lib. Etc.
- We’re going to leak:
- A .text pointer, so we can borrow a chunk of our program. This is great, because this is guaranteed not to change for our executable. Obviously :P
- A .stack pointer, so we can find our own array/buffer.
- A pointer to somewhere in libc, so we can use all the stuff in libc.

More types of shellcode

- Egghunter
  - If we only have a small space for our shellcode we can create an egghunter, which searches memory for a signature, and then jmp’s to it
- We then need to put our code into this other region, which is pretty easy to do in most apps
- May need to fix up memory permissions (especially in 2015) or map a new page, i.e. mprotect()

Omeletttt

- Omelet is a variation of an egghunter, in which instead of one big slap that contains our real payload, we have lots of little ones that either jump to each other, or are search for separately
- https://code.google.com/p/w32-seh-omelet-shellcode/

“A small piece of shellcode written in assembler that can scan the user-land address space for small blocks of memory ("eggs") and recombine the eggs into one large block. When done, the large block is executed. This is useful when you can only insert small blocks at random locations into a process and not one contiguous large block containing your shellcode in one piece: this code will recombine the eggs to create your shellcode in the process and execute it.”

Advanced payloads

- Userland exec
  - Executes a binary without using the kernel
  - Manually maps memory, loads the binary itself
  - Good for bypassing restrictions like SELINUX/apparmor/systrace, plus doesn't touch disk
  - https://grugq.github.io/docs/ul_exec.txt
- No public frameworks use it AFAIK, just priv8 onez

syscall proxy

- Syscalls are the only time you really interact with the operating system
- Run the majority of the shellcode locally but do all syscalls on the remote box
- Transfer all relevant state for the syscall, execute it, copy back the data
- Useful for a variety of reasons, chiefly antiforensics and ease of use (can run unmodified binaries remotely)

MOSDEF

- https://www.immunityinc.com/downloads/MOSDEF2.0.pdf
- A limited ‘c’ compiler but for the remote box
- Looks up relevant function addresses and objects
- Runs the code natively in the exploited process
- Can write 1 payload for linux and then ‘compile’ it for whatever architecture

Example

```
 self.localfunctions["malloc"]=("c","""
           #import
"remote","kernel32.dll|GlobalAlloc" as
"GlobalAlloc"
           char * malloc(int size) {
               char * buf;
               buf=GlobalAlloc(0,size);
               return buf;
           }
       """)
```

This code ‘imports’ the remote function GlobalAlloc from kernel32.dll

Encoding shellocde

- Often there will be ‘bad’ characters that you can’t use in shellcode, e.g. 0x00 (NULL), 0x0a ‘\n’, 0x0d ‘\r’
- Often also, you will need to stick to a subset of the character ranges such as only printable
- Doing this by hand is a pita, so we use an encoder/decoder
- Decoder must still be hand written in assembly so that it doesn’t contain the bad characters
- Exploit frameworks have these
  - https://www.offensive-security.com/metasploit-unleashed/msfencode/

Sometimes you just get lucky, though

- Think about a return path (i.e. you can set things up either by manipulating variables or just through luck) so that you eventually (without providing or specifying any addresses), end up with a call EBP, or a call ESP, etc.
- If our shell-code is executable, this is a win, regardless of ASLR.

Some more homework reading

http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt