A Brief History of Exploitation

Devin Cook
Auburn University
CIS 4930 / CIS 5930
Offensive Computer Security
Spring 2014

Overview

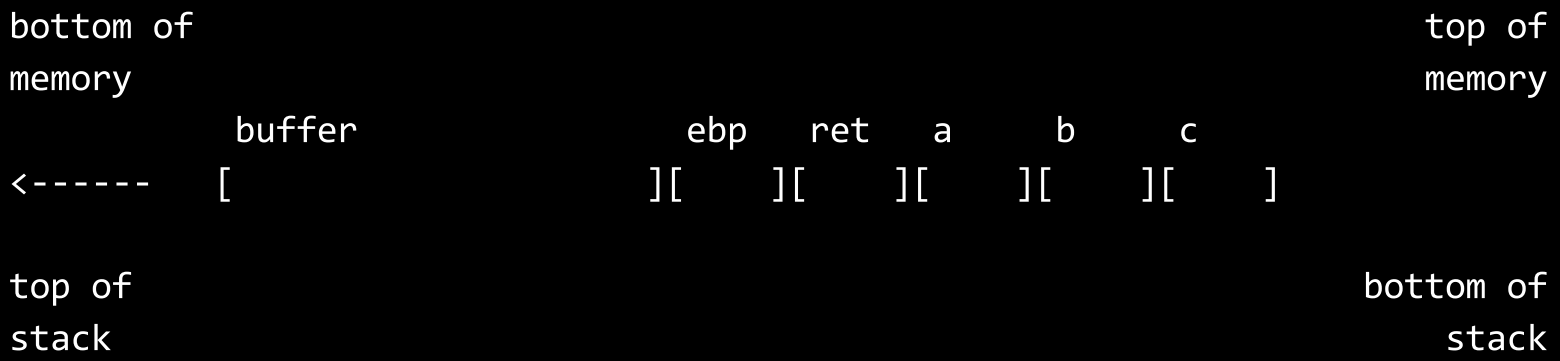
Security is a cat and mouse game. New mitigations are usually enacted as a reaction to an attack. Shortly after, new attacks are often developed that circumvent those mitigations.

- Stack canaries, SEH
  - overwrite variables instead
  - SEH hacking
- DEP/NX
  - ret2libc
  - ROP
- ASLR
  - brute force, address disclosure, lazy developers

Any other examples?

The Old Days

Things used to be so easy for us… Here's a stack frame:



The Old Days

We used to be able to just jump to shellcode via a ret instruction:

- Why is this possible?
- How can we prevent it?

Stack Canaries

What if we do this?

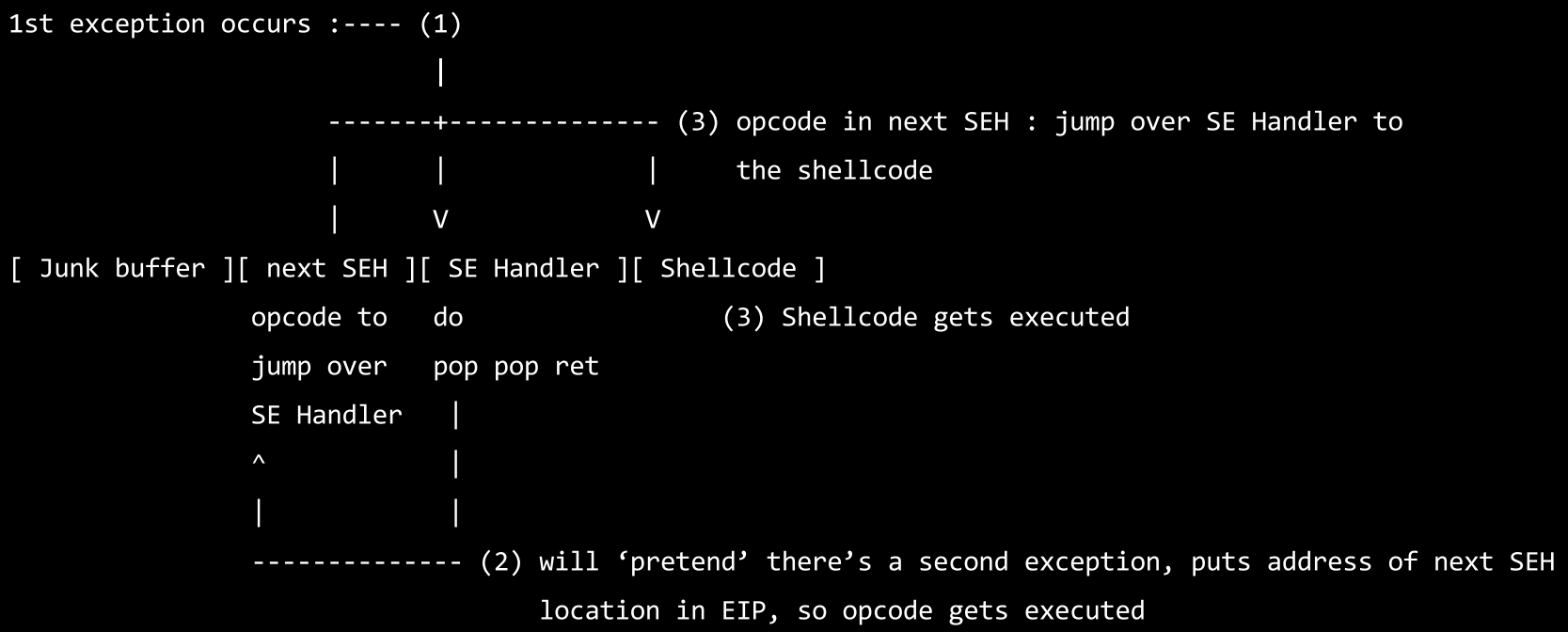
```
bottom of                                                    top of
memory                                                       memory
          buffer, locals    canary  ebp   ret   a     b     c
<------   [               ][0xbeef][    ][    ][    ][    ]

top of                                                    bottom of
stack                                                        stack
```

Now we can't easily overwrite the return address.

- could brute force, but not usually feasible
- better strategy is to smash local variables
  - maybe function pointers if we're lucky, otherwise maybe overwrite a pointer used for reading or writing memory so we can write to arbitrary locations
- SEH is a bit more interesting

Stack Exception Handler



If you can cause an exception to be triggered, you can run your shellcode. This is for Windows, but it’s kind of neat.

DEP/NX

Data execution prevention (DEP) prevents us from running anything stored on the stack, so that means shellcode is out.

Basically any writeable memory will not be marked as executable. If we can get control of EIP, though, can we still do anything with it?

DEP/NX

Data execution prevention (DEP) prevents us from running anything stored on the stack, so that means shellcode is out.

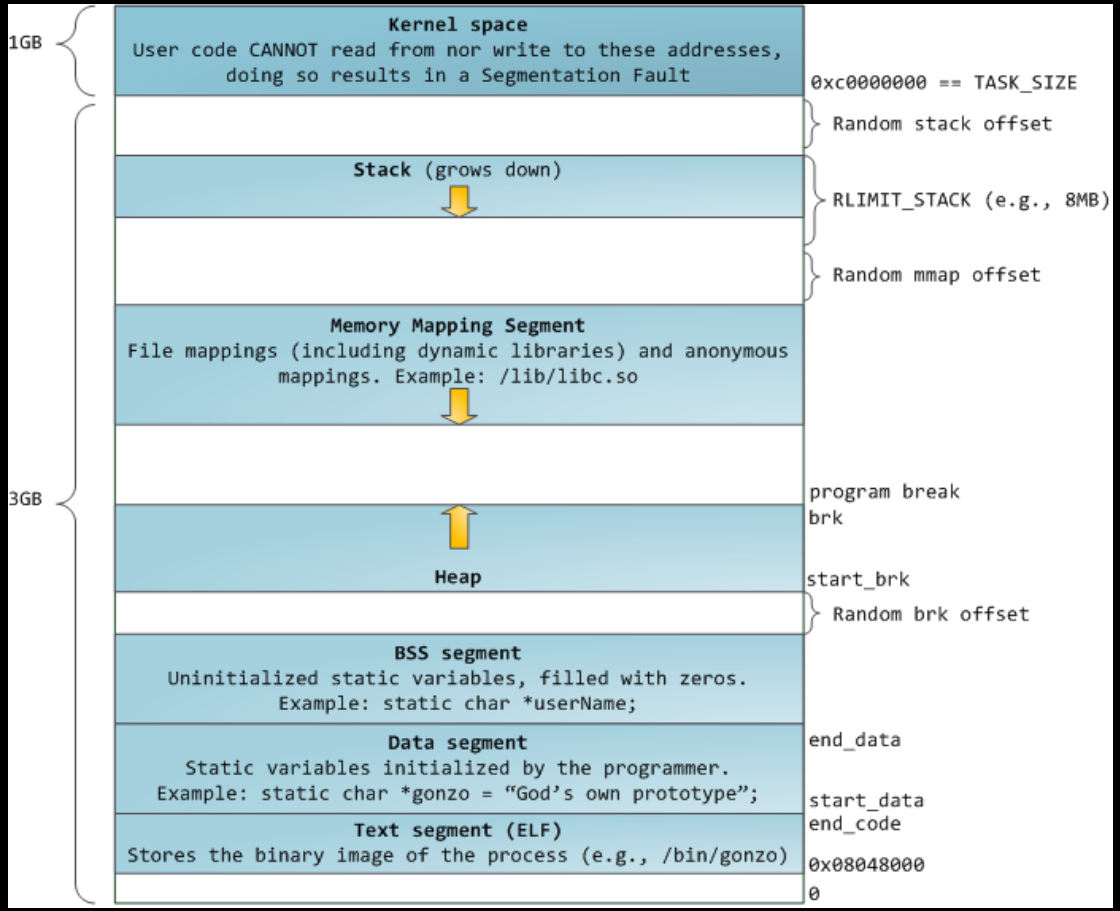
Basically any writeable memory will not be marked as executable. If we can get control of EIP, though, can we still do anything with it?

YEP

DEP/NX

Here's what the memory space of a process looks like.

See anything useful?

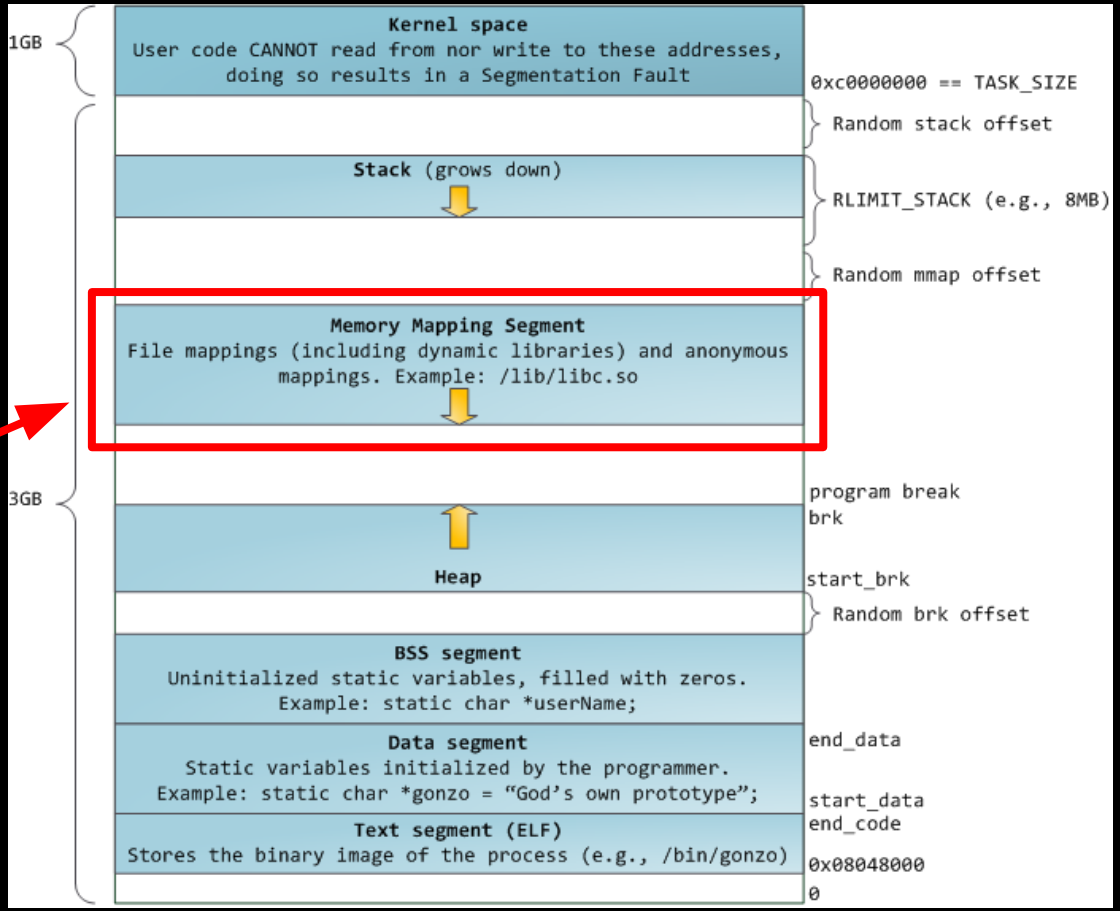


DEP/NX

Here's what the memory space of a process looks like.

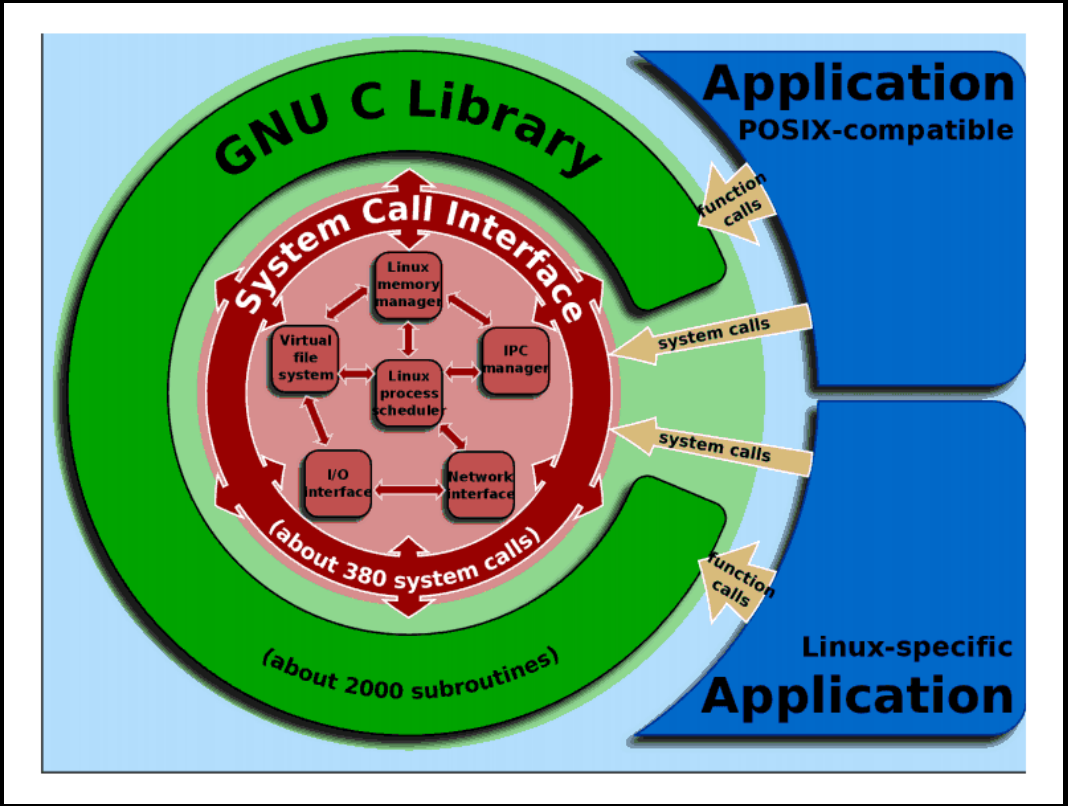
See anything useful?

OH BOY



ret2libc

2,000 subroutines? We can work with that. We just need to make it return into one of those.



ret2libc

This is pretty easy as long as the function isn't expecting any arguments or the arguments don't matter. Here's what it ends up looking like:

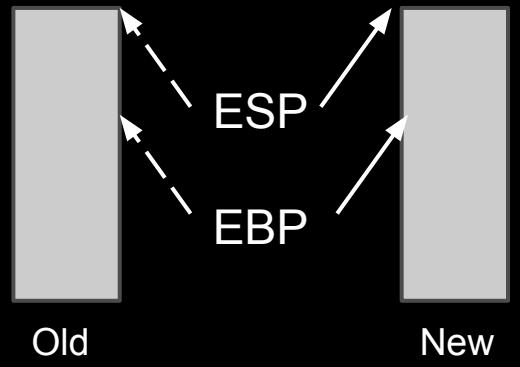
Unfortunately, to do anything useful we usually need to pass those functions some arguments.

Stack Pivoting

The current frame pointer (EBP) and stack pointer (ESP) are pointing at the frame we've just smashed. In order to make the called function see any arguments, we have to do something called a stack pivot. This allows us to create a stack with our own fake arguments and return addresses.

Basically we can create a new stack somewhere (maybe even on top of the old stack) and then fix the pointers.

How can we do that?



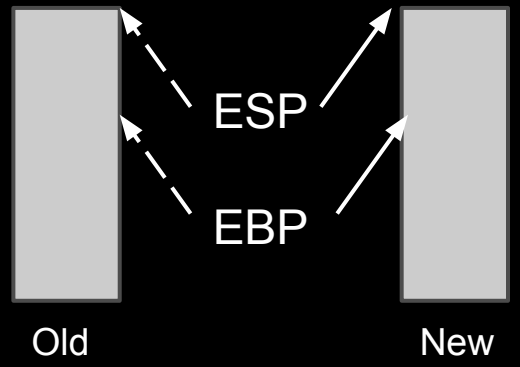
Stack Pivoting

The current frame pointer (EBP) and stack pointer (ESP) are pointing at the frame we've just smashed. In order to make the called function see any arguments, we have to do something called a stack pivot. This allows us to create a stack with our own fake arguments and return addresses.

Basically we can create a new stack somewhere (maybe even on top of the old stack) and then fix the pointers.

How can we do that?

- xchg registerContainingFakeStackAddress, ESP
- add ESP, SomeConstant
- sub ESP, SomeConstant
- mov ESP, registerContainingFakeStackAddress
- hack the function prologue because they modify ESP there
- hack the function epilogue because they modify ESP there



aside: ret2libc Mitigations

- We can remove unnecessary functions from the libraries we link.
  - That still leaves all the functions available in our own code, and all the library functions that we don’t remove.

- We can also make sure that the libraries get loaded into an area of memory with null bytes in the address.
  - Doesn’t help if we can write null bytes in our buffer overflow (i.e. not exploiting a string function).

Of course, making less functions available doesn’t prevent us from arbitrarily stringing together pieces of the available code in an order we specify...

ROP

Those techniques can be done with Return Oriented Programming (ROP). It utilizes small pieces of code, followed by a ret instruction. You can chain these “gadgets” together by simply writing their addresses in a row. This chain is often called a “ROP chain”.

Example gadgets:

```
push EAX
ret

sub EAX, 4
ret

pop EBX
xor EAX, EAX
ret
```

```
pop ECX
ret

add ECX, 8
ret

pop EAX
pop EBX
ret
```

ROP

Using ROP, we can accomplish several different things.

As Owen mentioned last week, if you have the right combination of gadgets, you can do ANYTHING (they can be Turing Complete).

Often the goal is to make some area of memory where your shellcode lives executable. You can also do things like change file descriptors.

You end up jumping around all over the memory space of the process, executing various bits of code that incrementally get you closer to the state you want it to be in.

ROP

Imagine the following gadgets:

- `xor %eax,%eax; ret` `0x08041111`
- `inc %eax; ret` `0x08042222`
- `pop %ebx; pop %ecx; ret` `0x08043333`

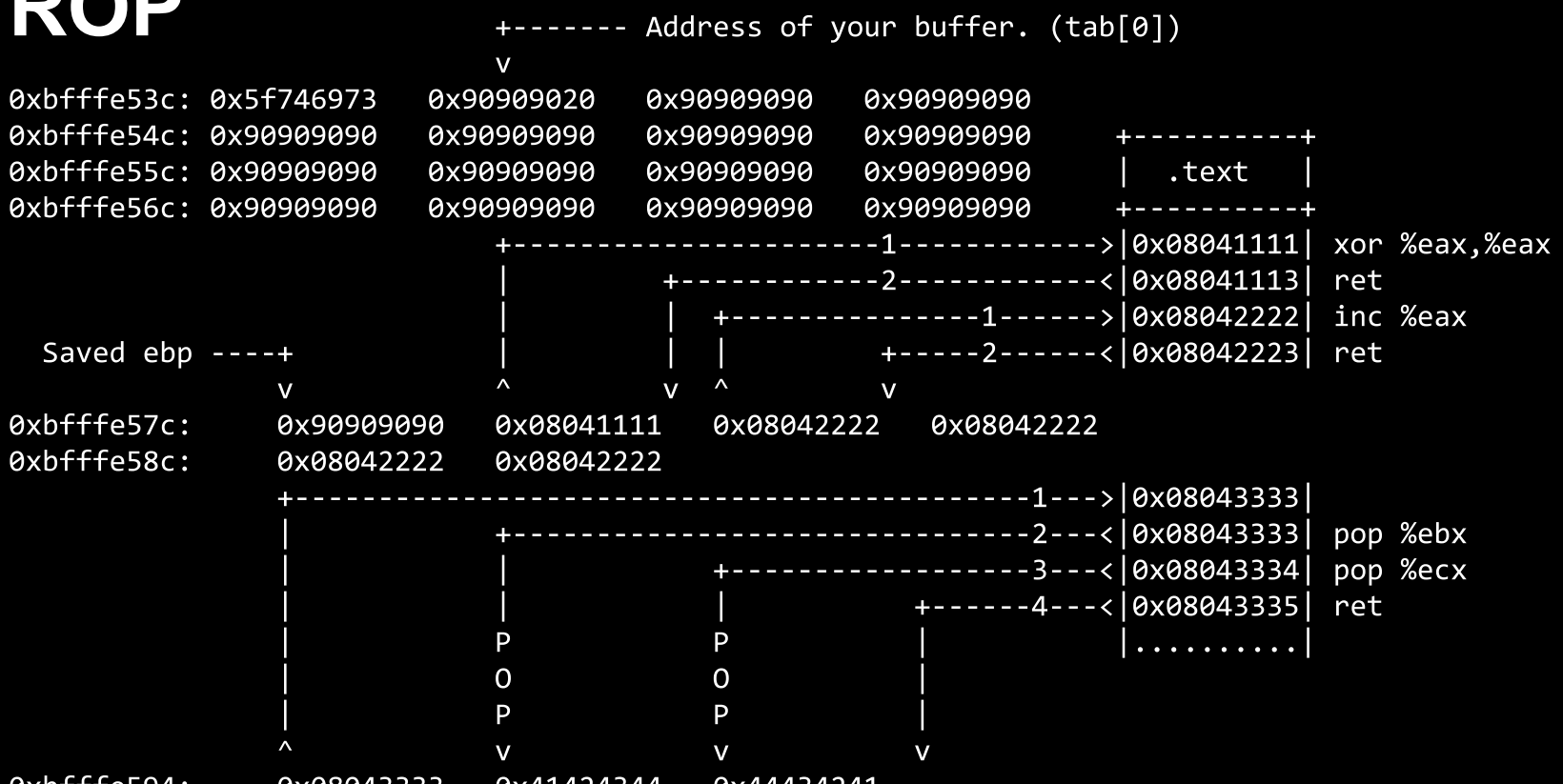
Before the buffer overflow the registers are:

- `%eax 0xbfffff53`
- `%ebx 0x080482a3`
- `%ecx 0x13`

The objective is to set the following registers:

- `%eax 0x4`
- `%ebx 0x41424344`
- `%ecx 0x44434241`

ROP


+------- Address of your buffer. (tab[0])
v
0xbfffe53c: 0x5f746973 0x90909020 0x90909090 0x90909090
0xbfffe54c: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffe55c: 0x90909090 0x90909090 0x90909090 0x90909090
0xbfffe56c: 0x90909090 0x90909090 0x90909090 0x90909090
.text
0x08041111 xor %eax,%eax
0x08041113 ret
0x08042222 inc %eax
0x08042223 ret
Saved ebp
0xbfffe57c: 0x90909090 0x08041111 0x08042222 0x08042222
0xbfffe58c: 0x08042222 0x08042222
0x08043333
0x08043333 pop %ebx
0x08043334 pop %ecx
0x08043335 ret
POP
0xbfffe594: 0x08043333 0x41424344 0x44434241

ROP

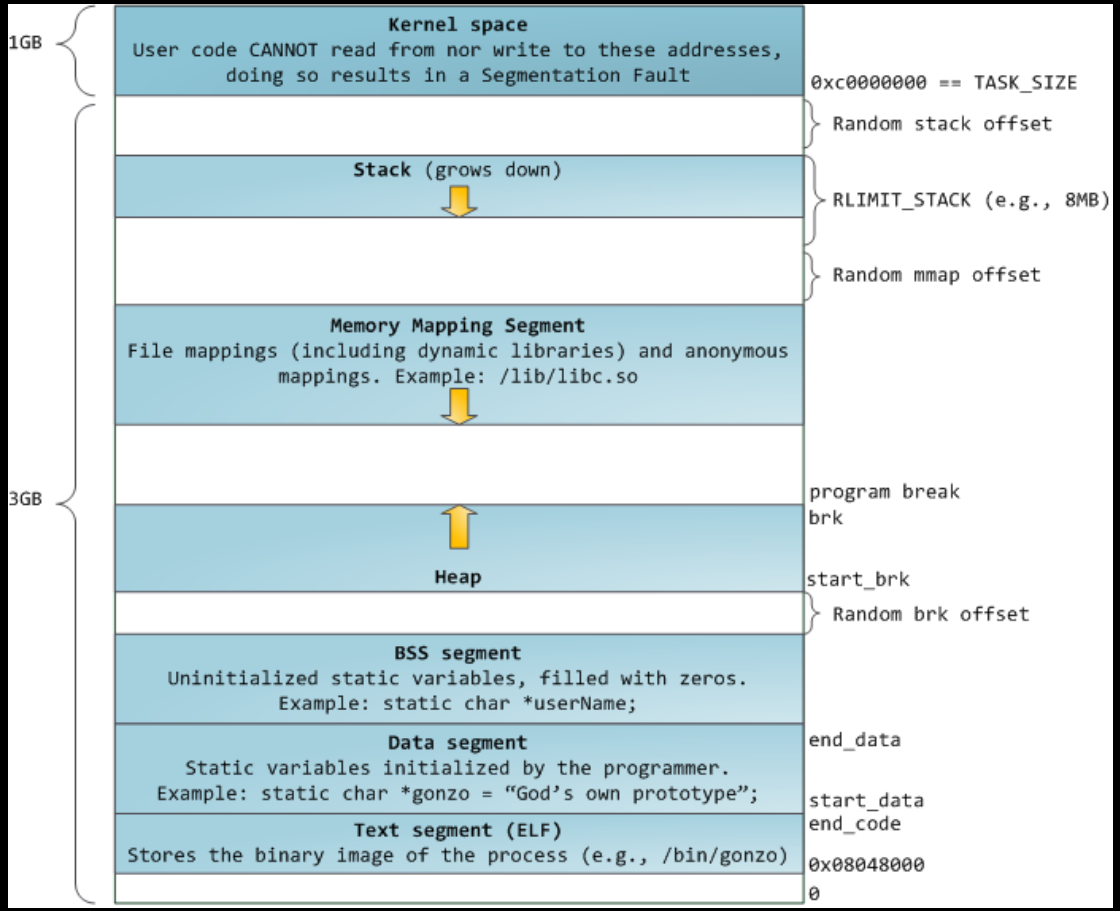
How can we make this more difficult?

ASLR

Address Space Layout Randomization changes the locations of everything in memory on a process-by-process basis.

32-bit: only 16 bits of entropy
64-bit: much more

important to note that ASLR is not always enabled for all libraries



Defeating ASLR

32-bit system: We can often brute force addresses.

If we don’t want to brute force, or we’re on a 64-bit system, we need some way of determining the address. This is called Address Disclosure.

Alternatively, there are some libraries that are still ASLR-disabled. That makes them pretty easy targets.

Fairly recently: http://cyvera.com/aslr-disabled-dropbox/

Future Mitigations

Control Flow Integrity

- Works by restricting where you can return to by various means
  - shadow call stack
  - access control for memory regions
- Still fairly academic, but looks promising

Sandboxing

- Used in Chrome, Adobe Reader, etc.
- Works pretty well, successful attacks must also escape the sandbox

Security is about increasing the cost (time and money) of an attack.

APT/targeted attacks can usually find a way in.

Tools

mona.py

- Immunity plugin, now works on WinDBG
- Can automatically generate ROP exploits for you

vivisect

- Debugger from the kenshoto guys
- Mostly in python, easily scriptable

ROPgadget

- Great tool for creating your ROP catalogue

ropc

- ROP compiler! Owen talked about this last week.

Post Exploitation

Once you have arbitrary code execution, what can you do with it?

- Read/Write files - open(), fread(), fwrite()
- Change user - setuid(), seteuid()
- Execute other programs - exec*()
- Phone home - socket(), connect(), nc
- Establish persistence
- Privilege Escalation
- etc...

Practice

War games and CTFs can be a great learning experience. They're also a great place to try out all of these techniques (sometimes on really weird architectures).

- http://repo.shell-storm.org/CTF/
- http://net-force.nl/challenges/
- http://microcorruption.com/
- http://io.smashthestack.org:84/

Questions?

References

- https://www.soldierx.com/tutorials/Stack-Smashing-Modern-Linux-System
- http://www.phrack.org/issues.html?issue=49&id=14
- http://www.phrack.org/issues.html?issue=58&id=4
- http://neilscomputerblog.blogspot.com/2012/06/stack-pivoting.html
- http://neilscomputerblog.blogspot.com/2013/04/rop-return-oriented-programming.html
- https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/
- http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/
- http://shell-storm.org/blog/Return-Oriented-Programming-and-ROPgadget-tool/