WINDOWS EXPLOITATION 101

Dino Dai Zovi / @dinodaizovi / ddz@theta44.org

Memory Corruption

- Memory corruption is when a programming error causes a program to access memory in an invalid way
  - Overwriting memory reserved for a different variable
  - Overwriting memory reserved for programming language runtime control structures
  - Access uninitialized or freed memory
- When memory corruption may allow an attacker to take control of a program, it is a security vulnerability

Memory Corruption Classes

* Buffer overflows (Stack, Heap, Data segment, etc)
* Format string injection
* Out-of-bounds array accesses
* Integer overflows (can lead to buffer overflows or out-of-bounds array access)
* Uninitialized memory use
* Dangling/stale pointers (i.e. use-after-free)

Memory Corruption Exploits

- Usually the goal is to inject a machine code payload ("shellcode") and get the target program to run it
    - Usually we just want it to give us a remote or higher-privileged shell (/bin/sh or cmd.exe)
    - Not all exploits will use a payload that runs a shell
- Not all memory corruption exploits execute shellcode

Solaris TTYPROMPT Bug

```
% telnet
telnet> environ define TTYPROMPT abcdef
telnet> o localhost

SunOS 5.8

bin c c c c c c c c c c c c c c c c c c c c c c c c c c c c c
c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c
c c c c\n
Last login: whenever
$ whoami
bin
```

Vulnerability Analysis

- A program crashes, is it repeatable and reproducible?
- Memory is corrupted, is it controllable?
- Memory corruption can be controlled, is it exploitable?
- Some tools are available to help
  - !exploitable (WinDbg)
  - Crash Wrangler (Mac OS X)

Exploit Development

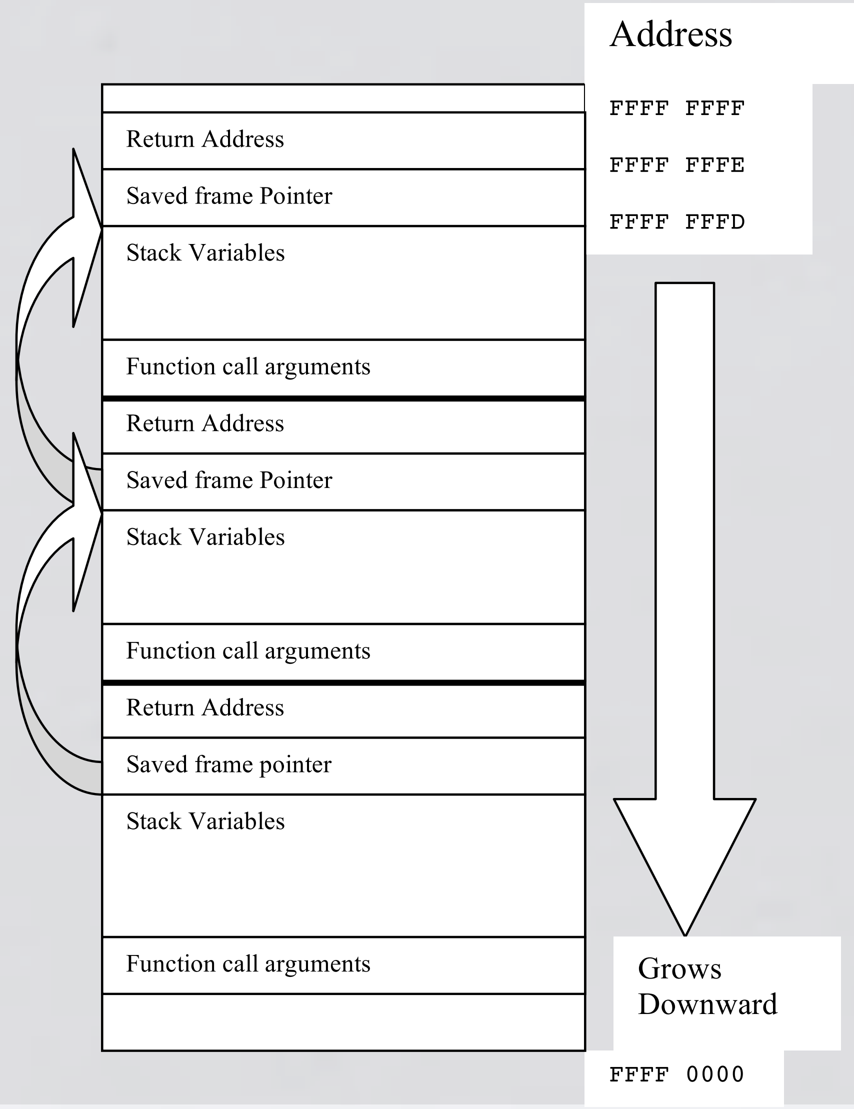
- Identify methods of controlling memory corruption
- Leverage controlled memory corruption to affect the program's behavior in a way that would give an attacker more privileges, capabilities, or access to the system
- Ideally, we would like to make it execute our payload
- Everyone loves a remote root/SYSTEM shell

Stack Buffer Overflows

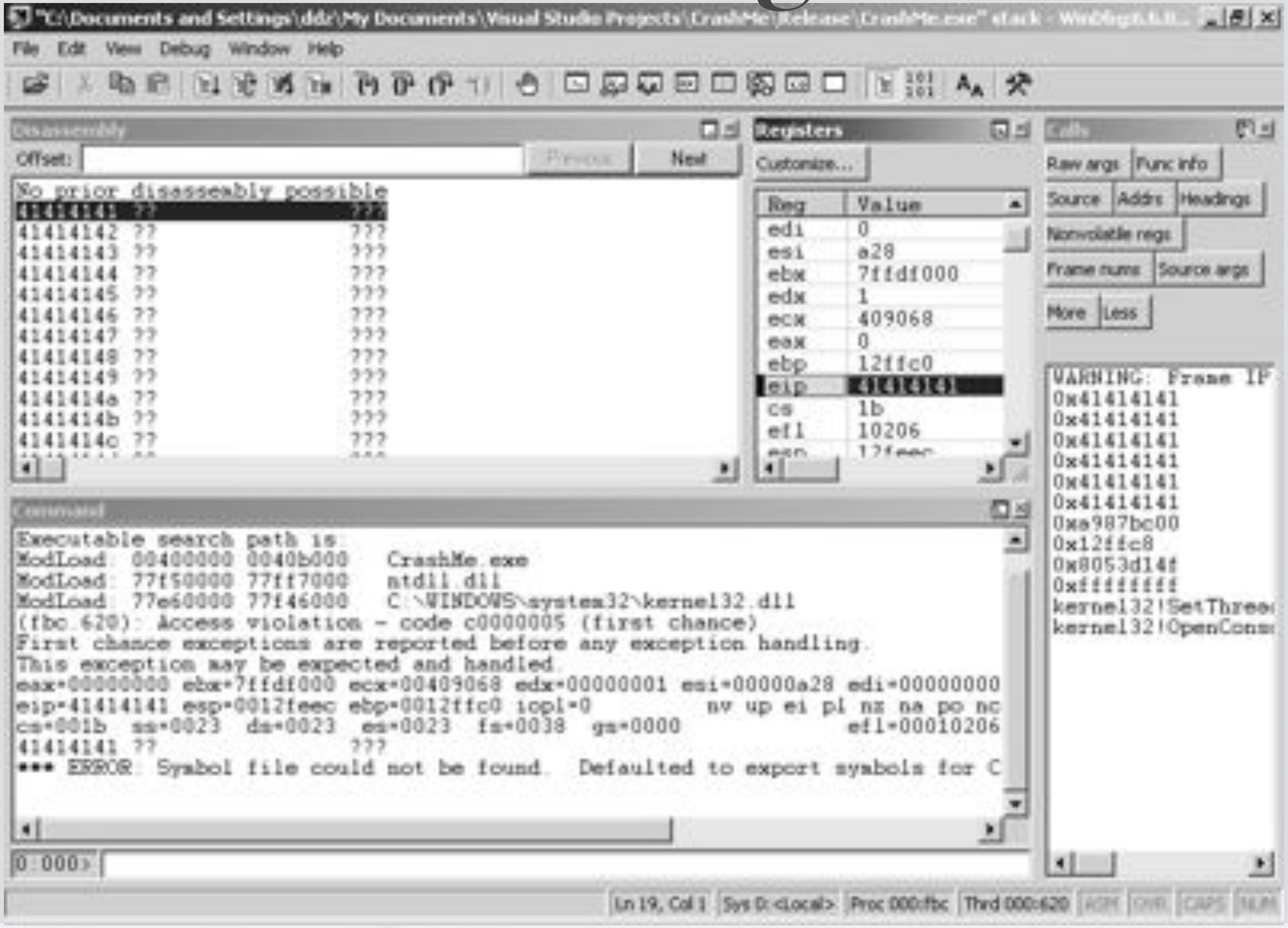
- The canonical, simplest type of memory corruption to understand and exploit
- First publicly used by Robert Morris worm in 1988
  - Used a stack buffer overflow in VAX BSD in.fingerd
- Are *still* exploitable on many systems today
  - Many operating systems and compilers include defenses against these now (more on this later)

The Stack

* Stack grows downward
* Memory writes go upward
* Stack variables can overflow into saved frame pointer and return address

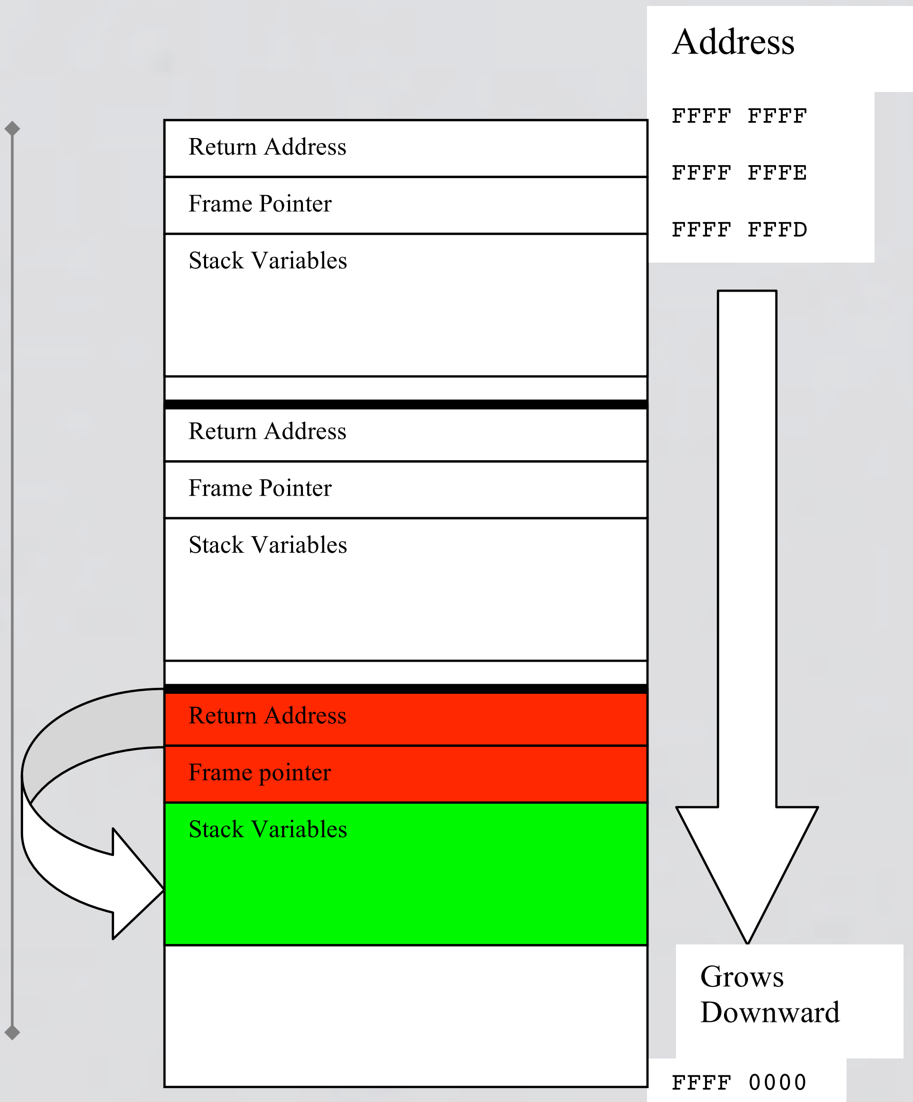


Smashing the Stack and controlling EIP



Stack Buffer Overflow

- Stack variable overflows, overwriting the return address
- The attacker writes a memory address in the stack for the return address
- The subroutine returns into payload on stack



LET’S SEE A REAL (FAKE) ONE…

Exploit By Numbers

1. Trigger the vulnerability
2. Identify usable characters for attack string
3. Identify offsets and significant elements in attack string
4. Fill in jump addresses, readable/writable addresses, etc
5. Identify amount of usable space for the payload
6. Drop in payload

Trigger the Vulnerability

- Write a network client to talk to the server
- Create a malformed file that gets opened by the app
  - Document (.doc, .ppt, .pdf)
  - Media file (.mp3, .mov, .wmv)
- Create a malicious web page that is viewed by browser
- **Cause the target application to crash**

Identify Usable Characters

- The *attack string* is the part of the input that triggers the vulnerability and contains values for overwritten memory (and possibly the payload also)
- Certain characters in the attack string may cause the application to parse the input differently and not trigger the vulnerability (*"bad bytes"*)
  - NULL bytes (any ASCII string)
  - Whitespace (\t\n\r)

Identify Offsets

- Use a *pattern string* to identify offsets into your attack string of data placed into registers or written to memory
- We are going to use Metasploit's **pattern_create.rb**

```
% pattern_create.rb 32
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab

% pattern_offset.rb 0x41366141
18
```

Fill in Memory Addresses

- For an exploit to function, certain parts of the attack string may need to readable, writable, or executable memory addresses
  - In particular, we want to overwrite the return address with the memory address of executable code
  - This memory address will redirect execution into our attack string
  - Spend quality time in your target's address space

Identify Usable Space

- We need to know how much room we have for our payload
- We will size it out by placing increasingly large numbers of NOPs followed by a debug interrupt (int 3)
- If the target generates a breakpoint exception, we have that much usable space
- If the target crashes in another way, we may need to shrink the payload space

Drop in Payload

- The payload must also not use any bad bytes or else it may get truncated and not execute properly
- For simple payloads and vulnerabilities, avoiding NULL bytes in the instruction encodings may be enough
- For more complex payloads and vulnerabilities, a payload decoder may be used to decode the payload before executing

LIVE DEMO TIME…

EXPLOITATION MITIGATION

Exploitation Mitigation

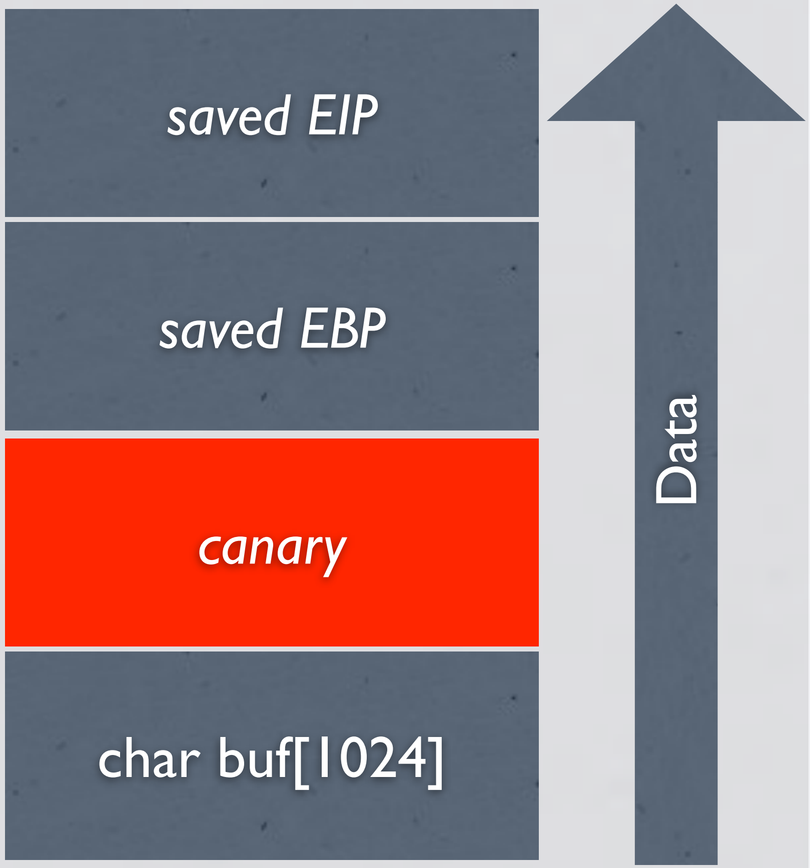
- Finding and fixing every vulnerability is impossible
- It is possible to make exploitation more difficult through:
  - Memory page protection
  - Run-time validation
  - Obfuscation and Randomization
- Making every vulnerability non-exploitable is impossible

Timeline of Mitigations

- Windows 1.0 - Windows XP SP1
  - Corruption of stack and heap metadata is possible
- Windows Server 2003 RTM
  - Operating System is compiled with stack cookies
- Windows XP SP 2
  - Stack/heap cookies, SafeSEH, Software/Hardware DEP
- Windows Vista
  - Address Space Layout Randomization

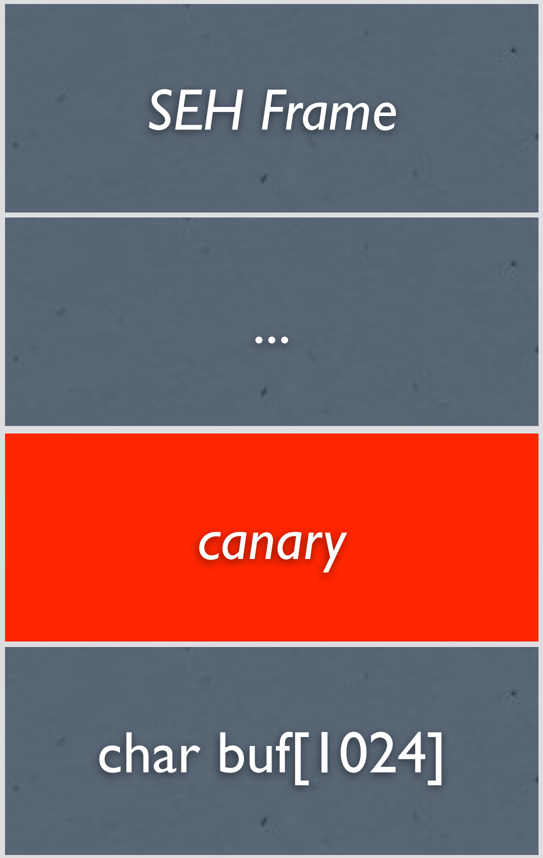
Visual Studio /GS Flag

- Place a random “cookie” in stack frame before frame pointer and return address
- Check cookie before using saved frame pointer and return address



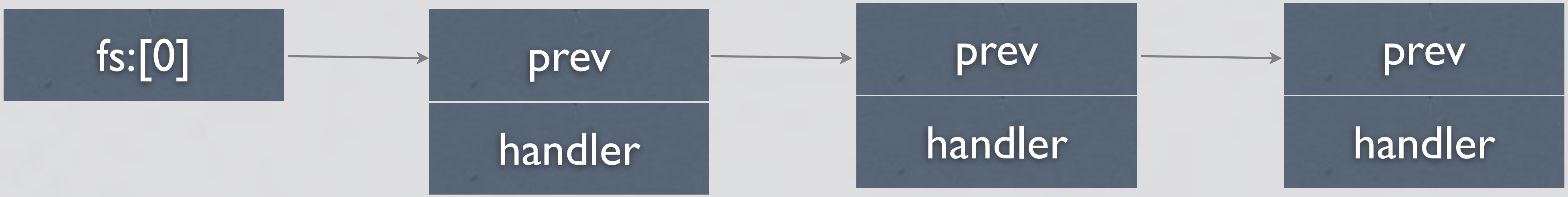
Structured Exception Handling

- Supports __try/__except blocks in C and C++ exceptions
- Nested SEH frames are stored on stack
- Contain pointer to next frame and exception filter *function pointer*



SEH Frame Overwrite Attack

- Overwrite an exception handler function pointer in SEH frame and cause an exception before any of the overwritten stack cookies are detected
    - i.e. run data off the top of the stack
- David Litchfield, "Defeating the Stack Based Buffer Overflow Protection Mechanism of Microsoft Windows 2003 Server"



Visual Studio /SafeSEH

- Pre-registers all exception handlers in the DLL or EXE
- When an exception occurs, Windows will examine the pre-registered table and only call the handler if it exists in the table
- What if one DLL wasn't compiled w/ SafeSEH?
  - Windows will allow any address in that module as an SEH handler
  - This allows an attacker to still gain full control

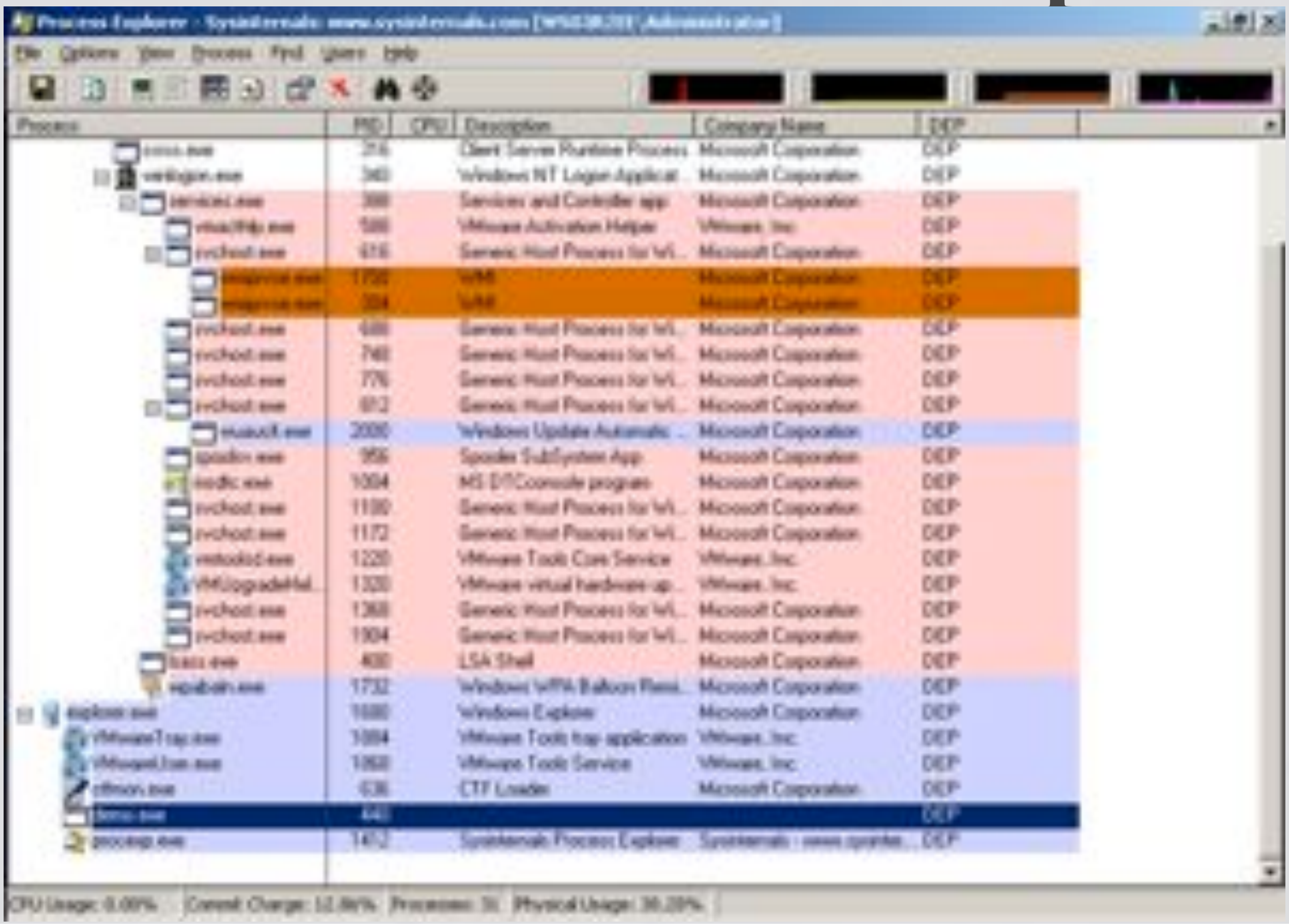
RTL Heap Safe Unlinking

* Corrupting the next/prev linked list pointers of a heap block on the free list allows an attacker to write a chosen value to a chosen location when that block is removed from the free list
    * i.e. Overwrite the global UnhandledExceptionFilter
* Safe Unlinking adds a 16-bit cookie to heap header, which is checked before the block is removed

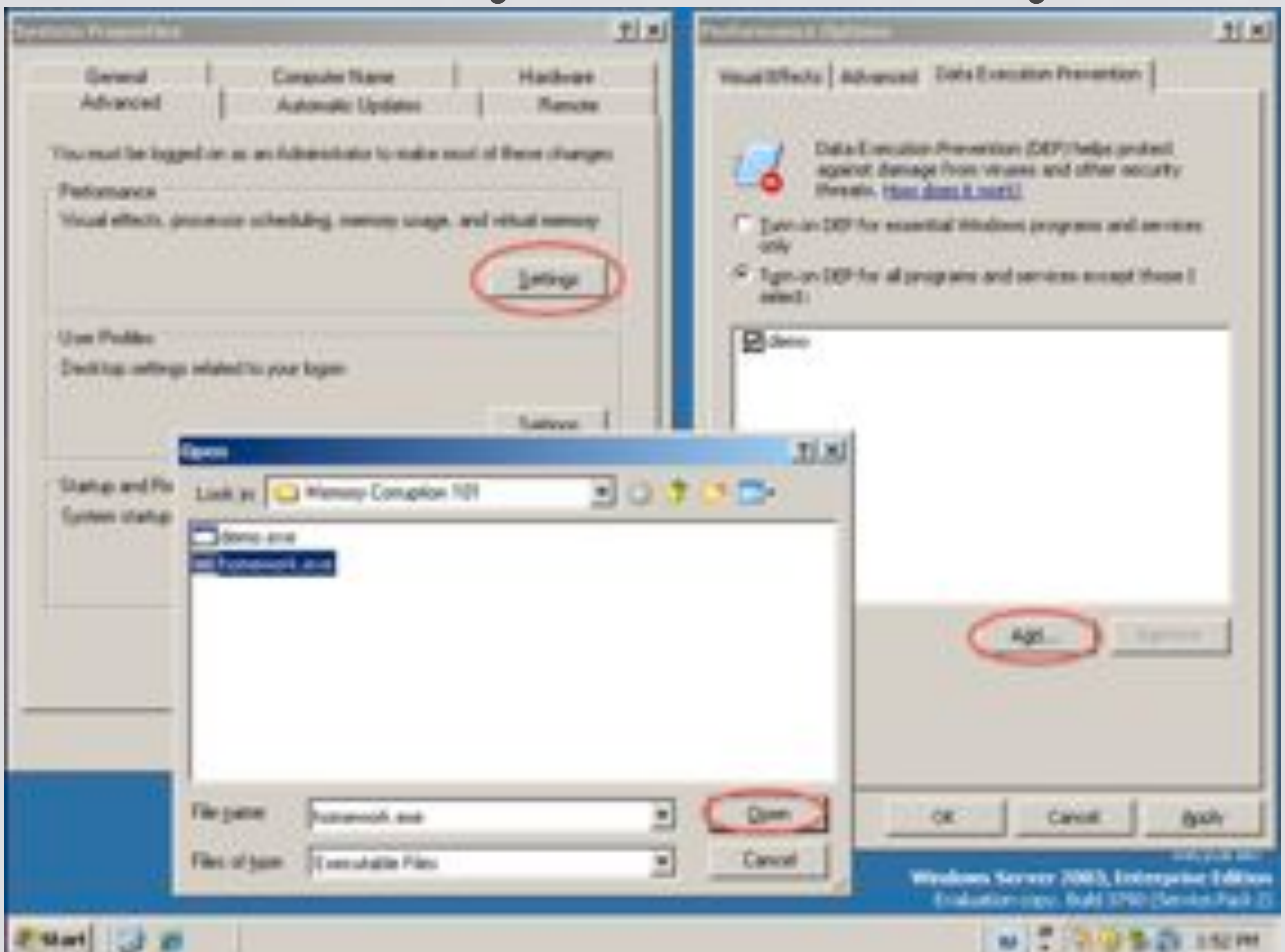
Data Execution Prevention

- Software DEP
  - Makes sure that SEH exception handlers point to non-writable memory (weak)
- Hardware DEP
  - Enforces that processor does not execute instructions from data memory pages (stack, heap)
  - Make page permission bits meaningful (R !=> X)

DEP Status in Process Explorer



Modify DEP Policy



Bypassing DEP

- Return-to-libc / code reuse
  - Return into the beginning of a library function
  - Function arguments come from attacker-controlled stack
  - Can be chained to call multiple functions in a row
- On XP SP2 and Windows 2003, attacker could return to a particular place in NTDLL and disable DEP for the entire process

WriteProcessMemory() DEP Evasion

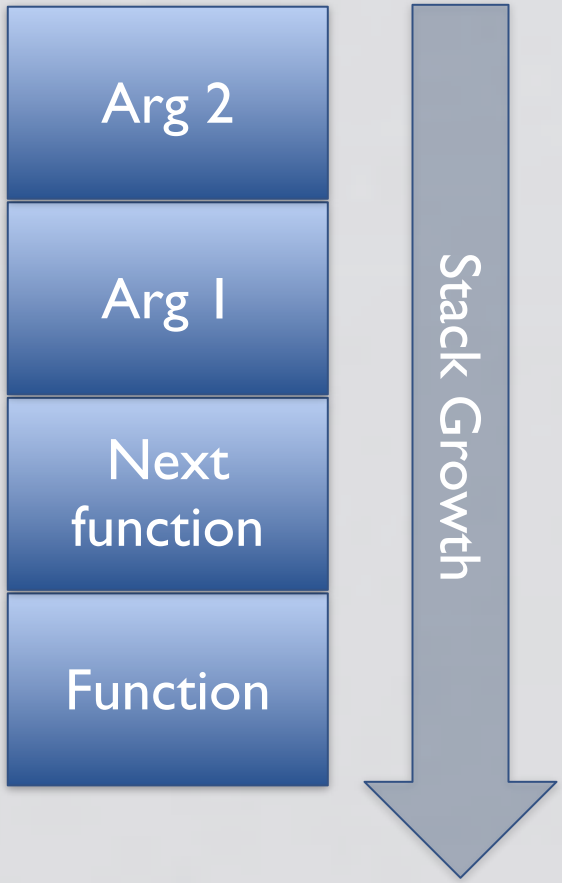
- Posted by Spencer Pratt to Full-Disclosure on 3/30[1]
- Return into WriteProcessMemory() function with crafted arguments so that it overwrites itself in memory
  - WPM() will bypass memory page permissions
  - Writes new code that executes right after WPM calls NtWriteVirtualMemory() returns
- Use WPM() to copy 1-3 byte chunks at known locations in memory together to form shellcode

1. "Clever DEP Trick", http://seclists.org/fulldisclosure/2010/Mar/553

RETURN-ORIENTED PROGRAMMING

Return-to-libc

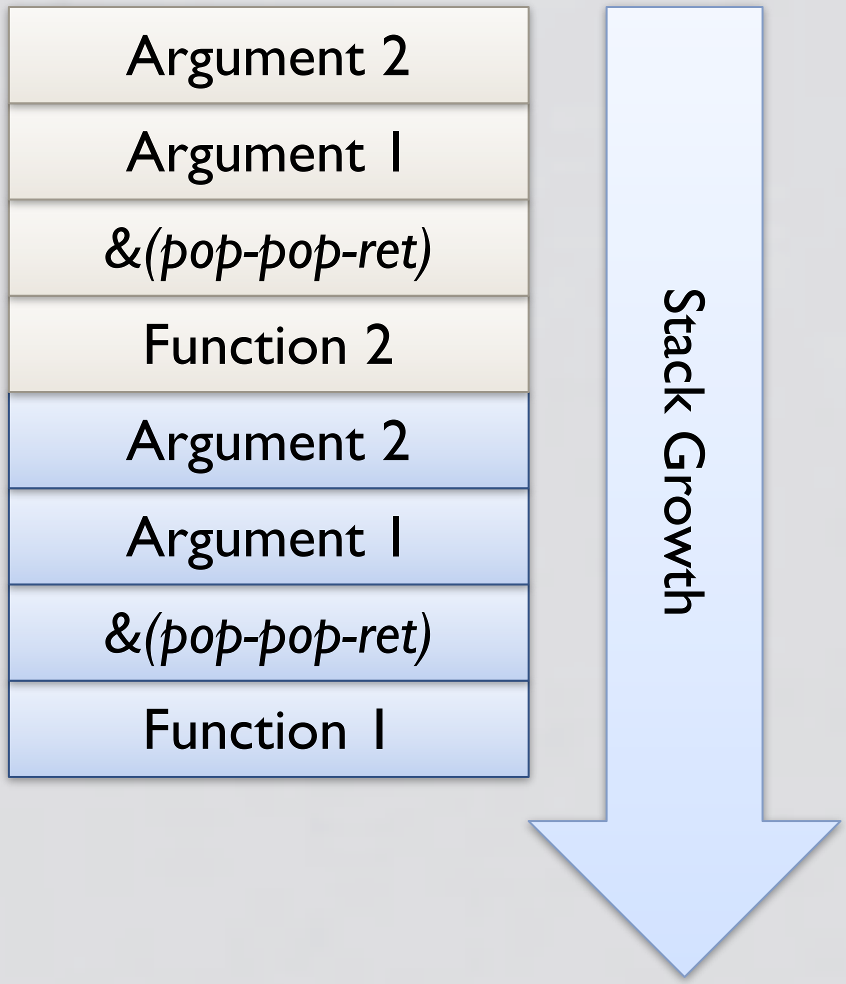
- Return-to-libc (ret2libc)
  - An attack against non-executable memory segments (DEP, W^X, etc)
  - Instead of overwriting return address to return into shellcode, return into a loaded library to simulate a function call
  - Data from attacker's controlled buffer on stack are used as the function's arguments
  - i.e. call system(*cmd*)



"Getting around non-executable stack (and fix)", Solar Designer (BUGTRAQ, August 1997)

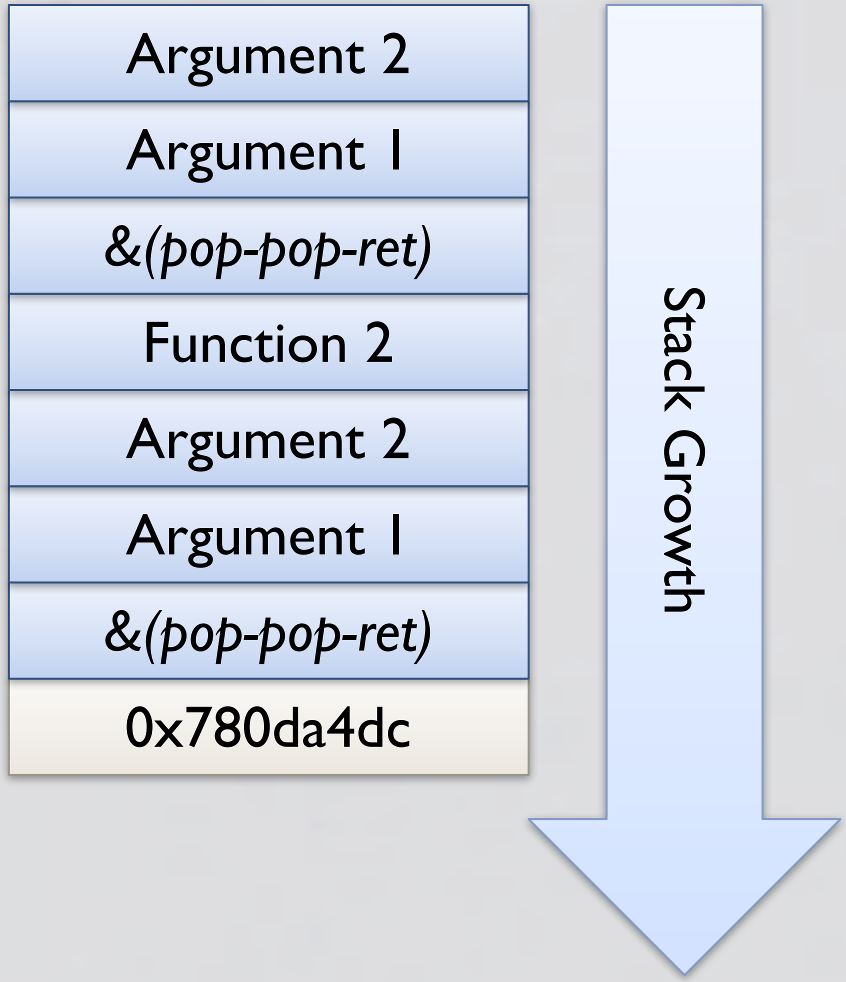
Return Chaining

- Stack unwinds upward
- Can be used to call multiple functions in succession
- First function must return into code to advance stack pointer over function arguments
  - i.e. pop-pop-ret
  - Assuming cdecl and 2 arguments



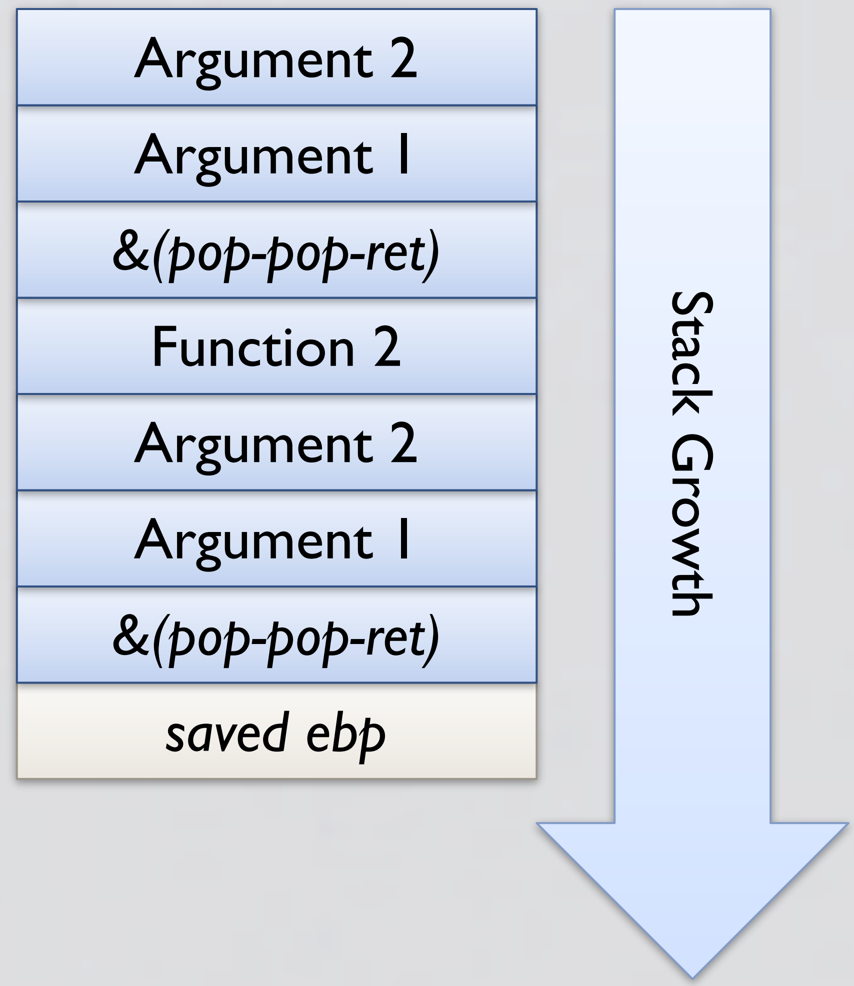
Return Chaining

```
0043a82f:

    ret

    …
```



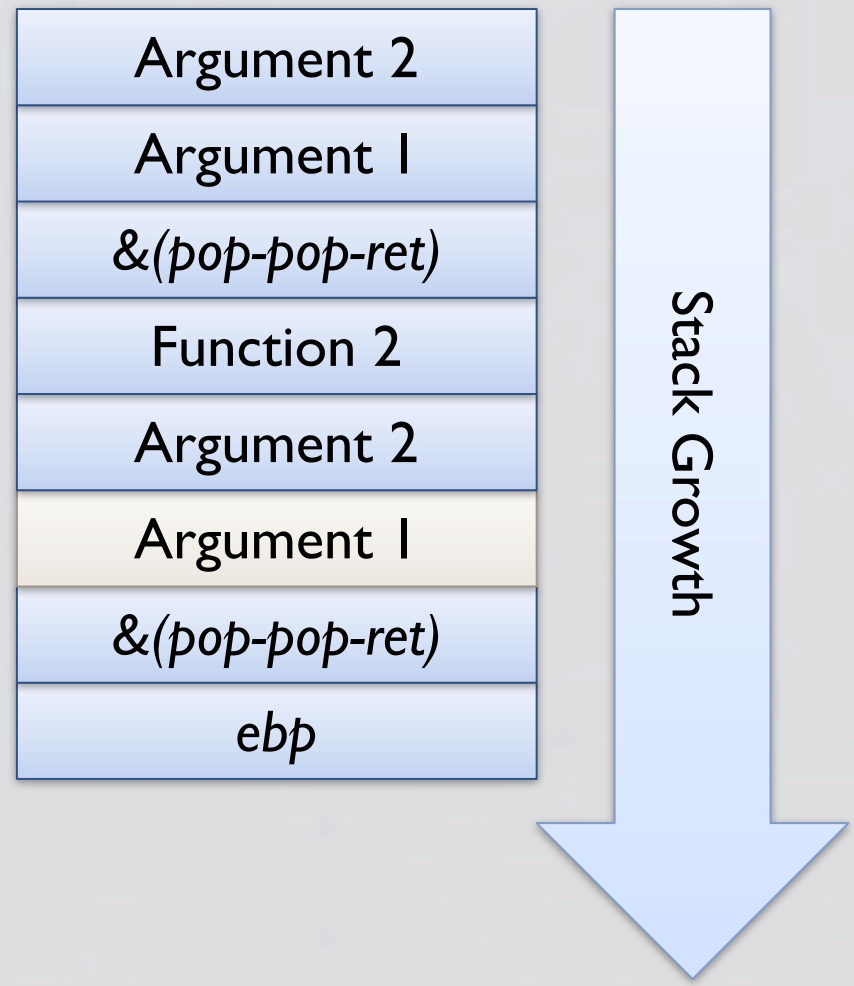
Return Chaining

```
780da4dc:
    push ebp
    mov ebp, esp
    sub esp, 0x100
    …
    mov eax, [ebp+8]
    …
    leave
    ret
```



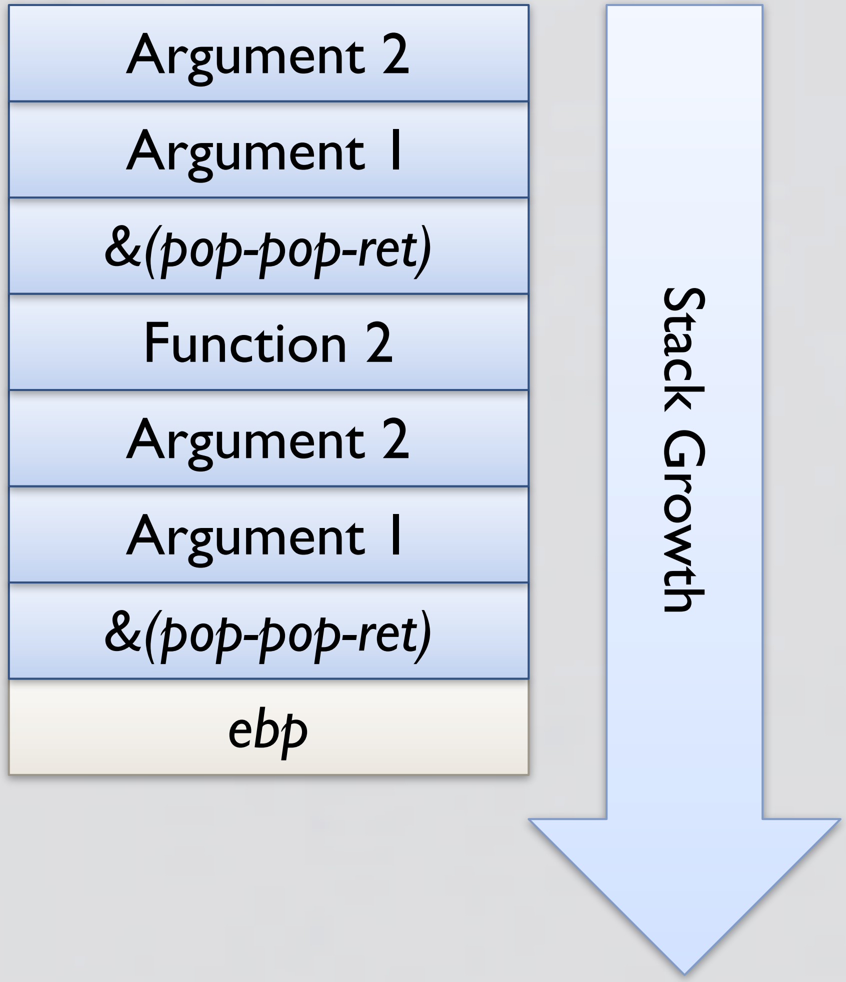
Return Chaining

```
780da4dc:
    push ebp
    mov ebp, esp
    sub esp, 0x100
    …
    mov eax, [ebp+8]
    …
    leave
    ret
```



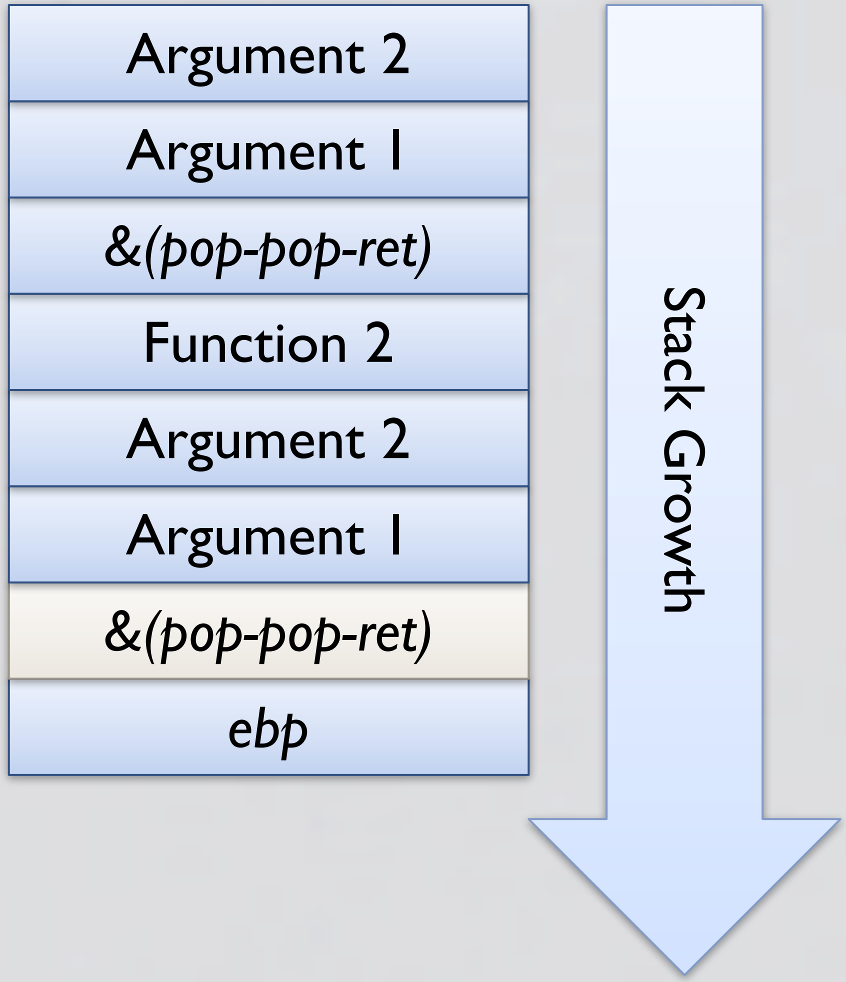
Return Chaining

```
780da4dc:
    push ebp
    mov ebp, esp
    sub esp, 0x100
    …
    mov eax, [ebp+8]
    …
    leave
    ret
```



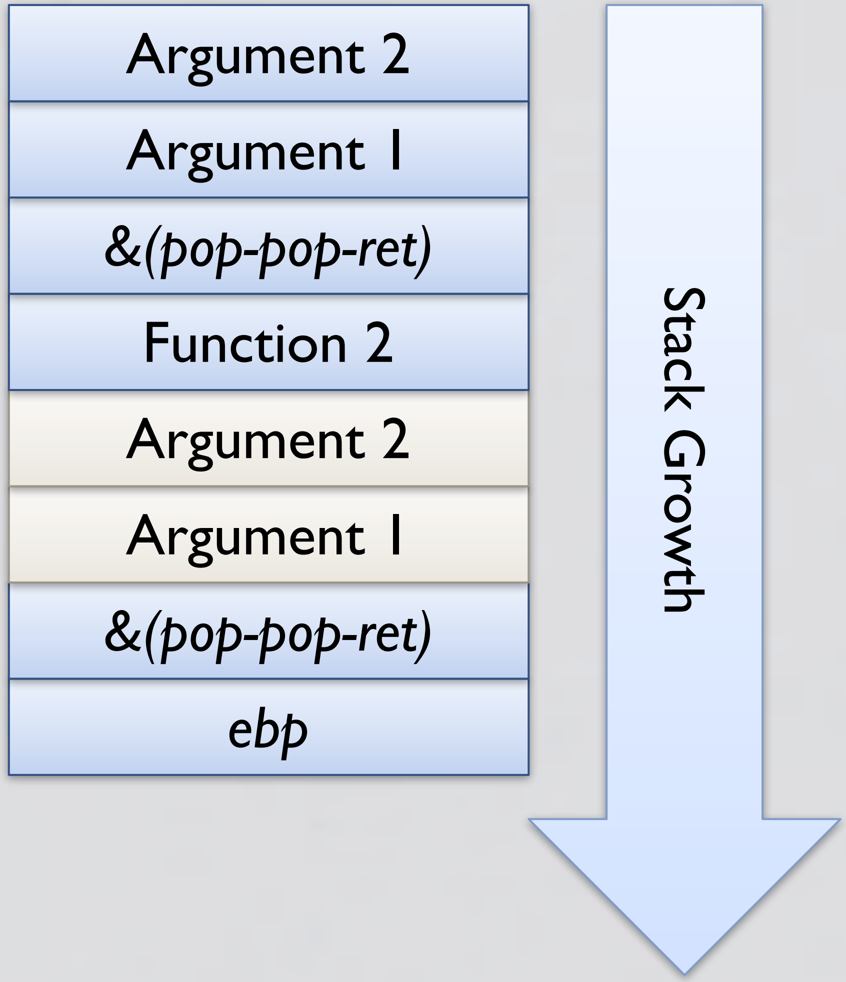
Return Chaining

```
780da4dc:
    push ebp
    mov ebp, esp
    sub esp, 0x100
    …
    mov eax, [ebp+8]
    …
    leave
    ret
```



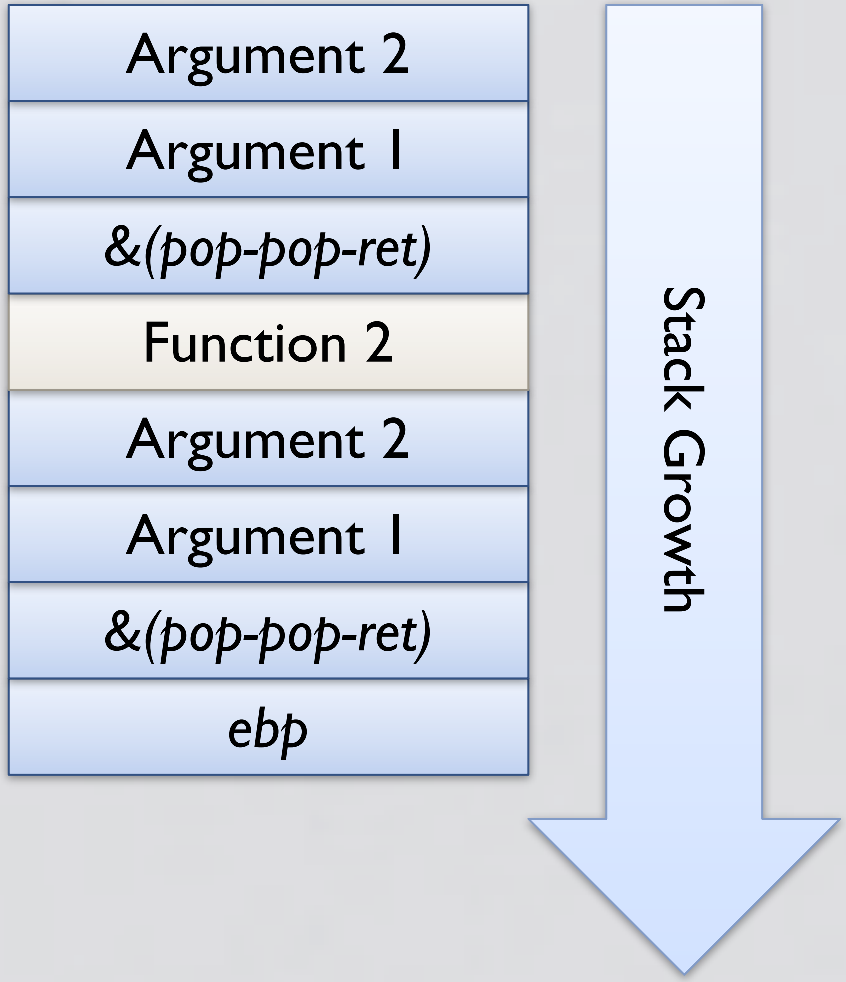
Return Chaining

```
6842e84f:
    pop edi
    pop ebp
    ret
```



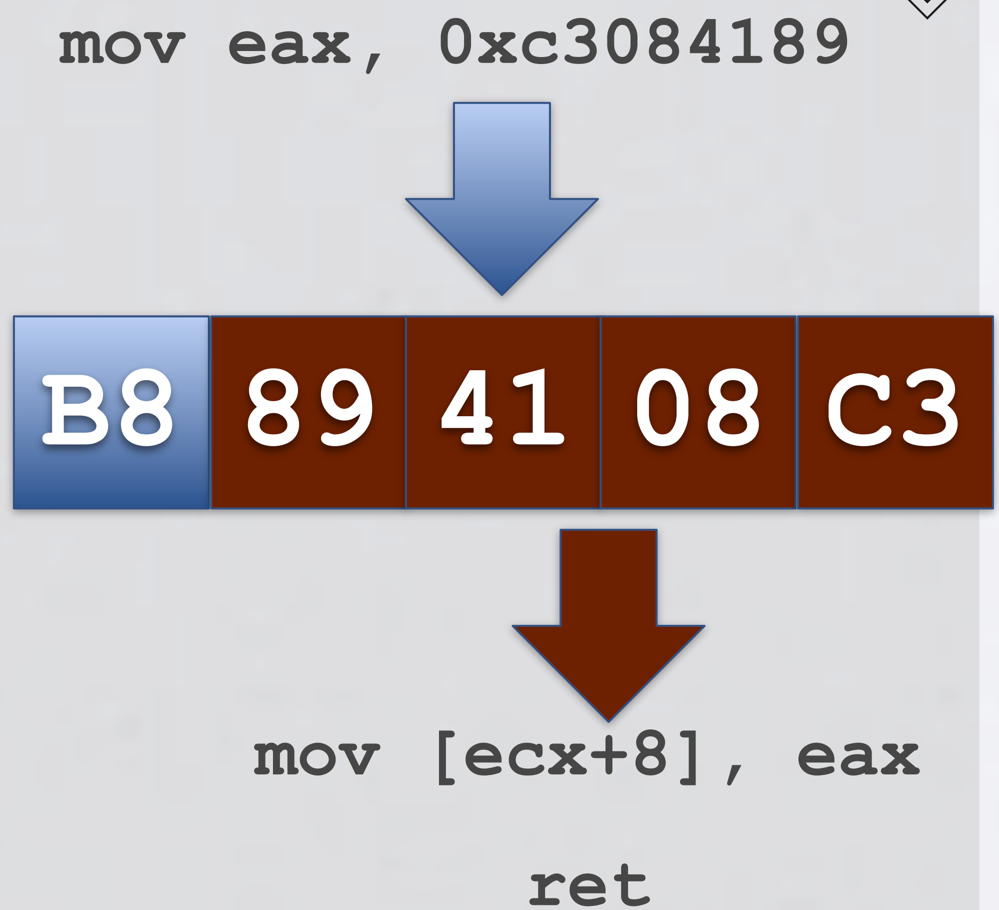
Return Chaining

```
6842e84f:
    pop edi
    pop ebp
    ret
```

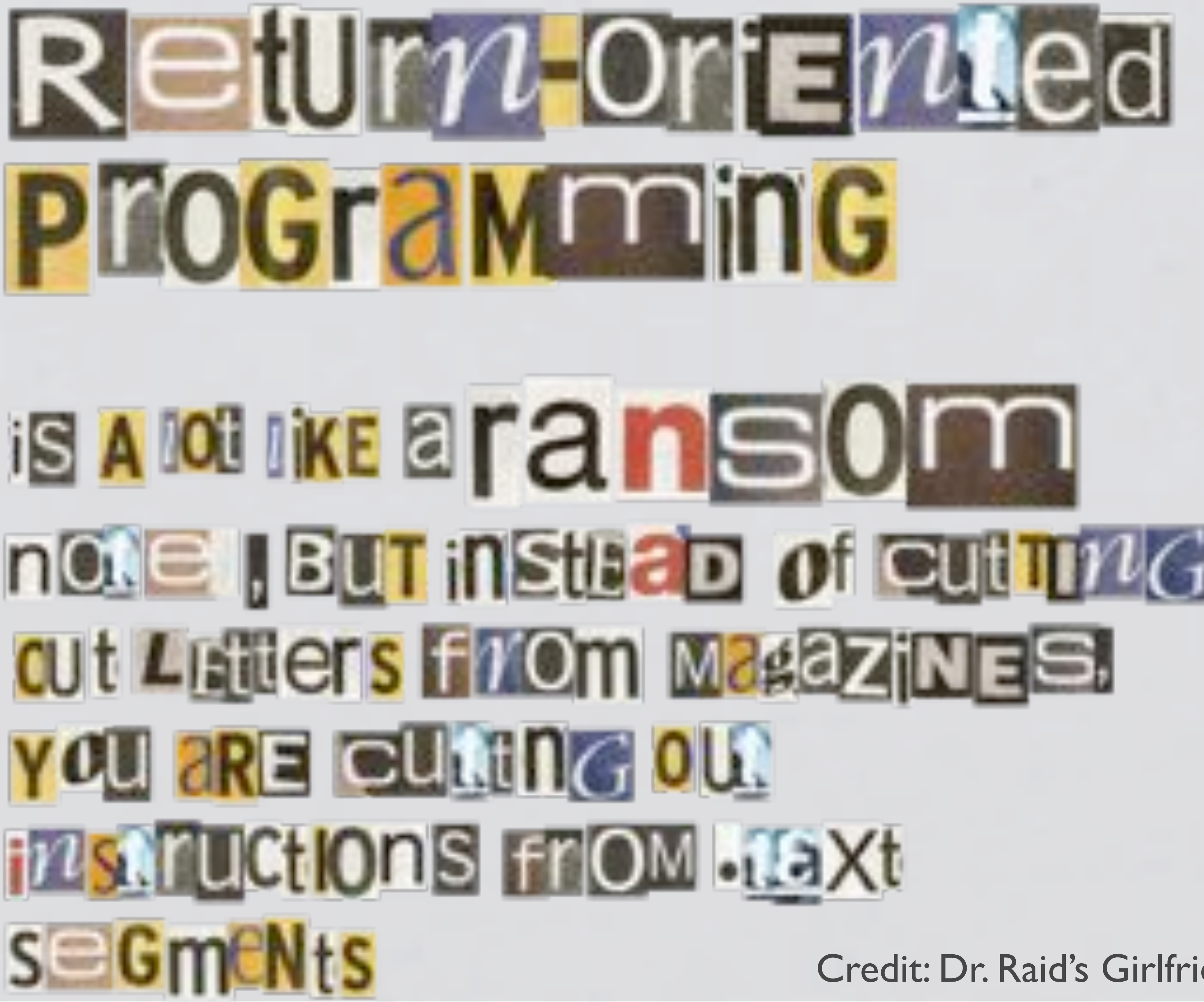


Return-Oriented Programming

- Instead of returning to functions, return to instruction sequences followed by a return instruction
- Can return into middle of existing instructions to simulate different instructions
- All we need are useable byte sequences anywhere in executable memory pages



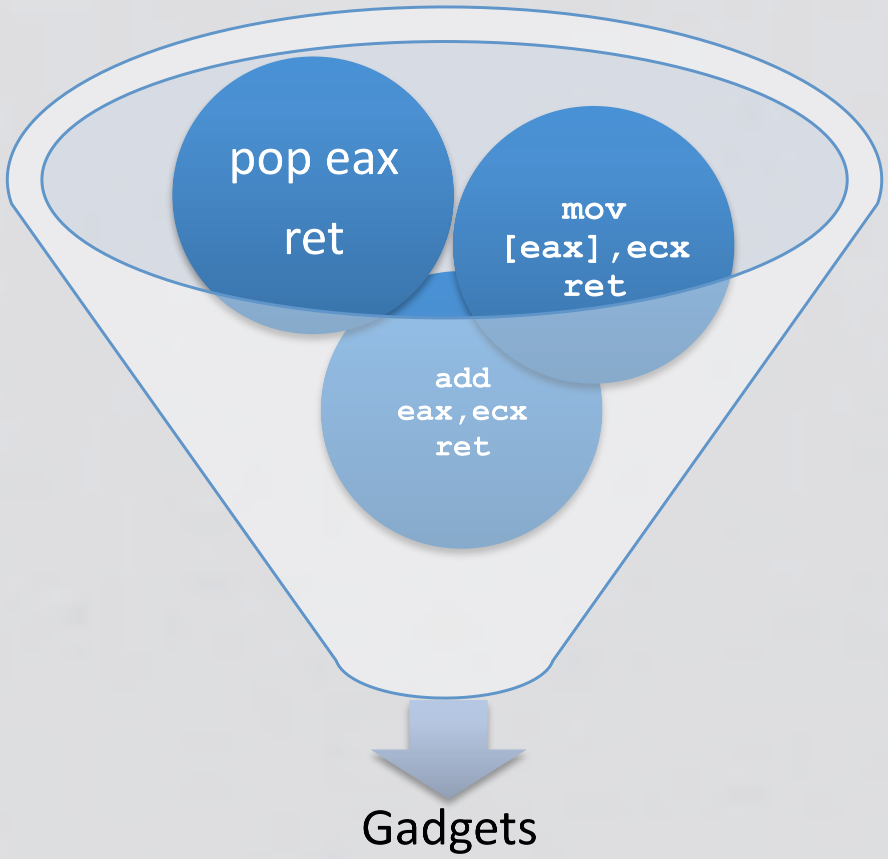
“The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86)”, Hovav Shacham (ACM CCS 2007)



Credit: Dr. Raid's Girlfriend

Return-Oriented Gadgets

- Various instruction sequences can be combined to form *gadgets*
- Gadgets perform higher-level actions
  - Write specific 32-bit value to specific memory location
  - Add/sub/and/or/xor value at memory location with immediate value
  - Call function in shared library



Example Gadget

```
pop eax
ret
```

+

```
pop ecx
ret
```

+

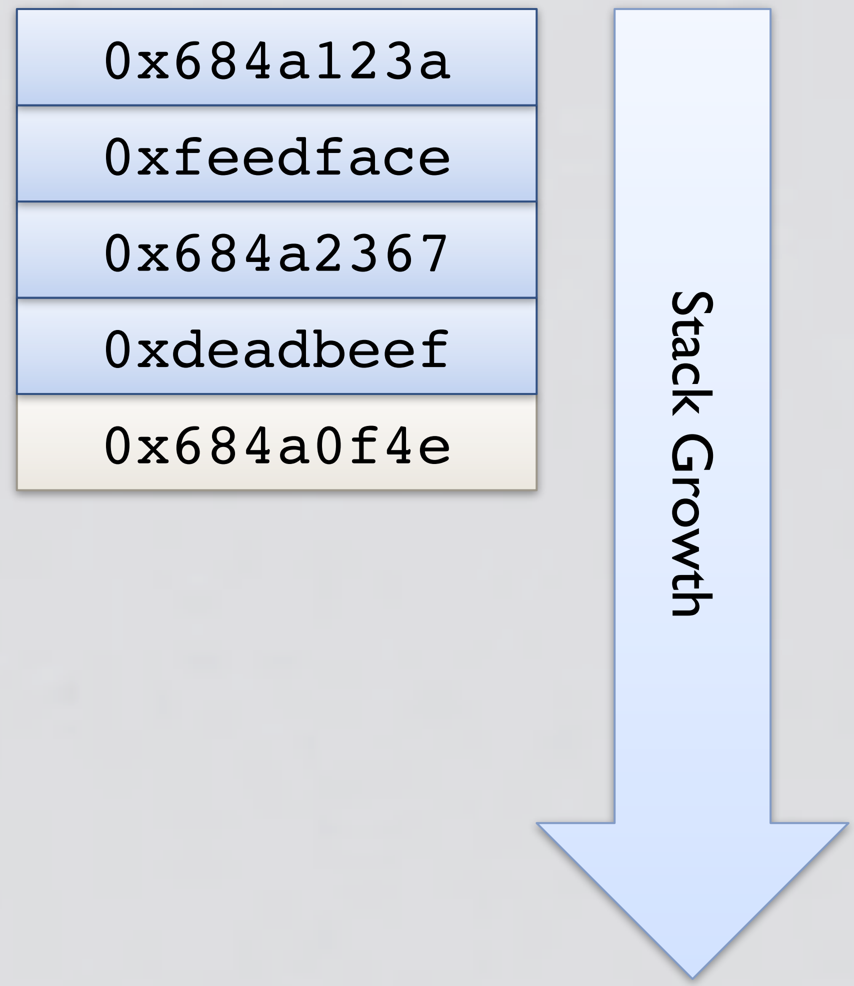
```
mov
[ecx],eax
ret
```

=

STORE IMMEDIATE VALUE

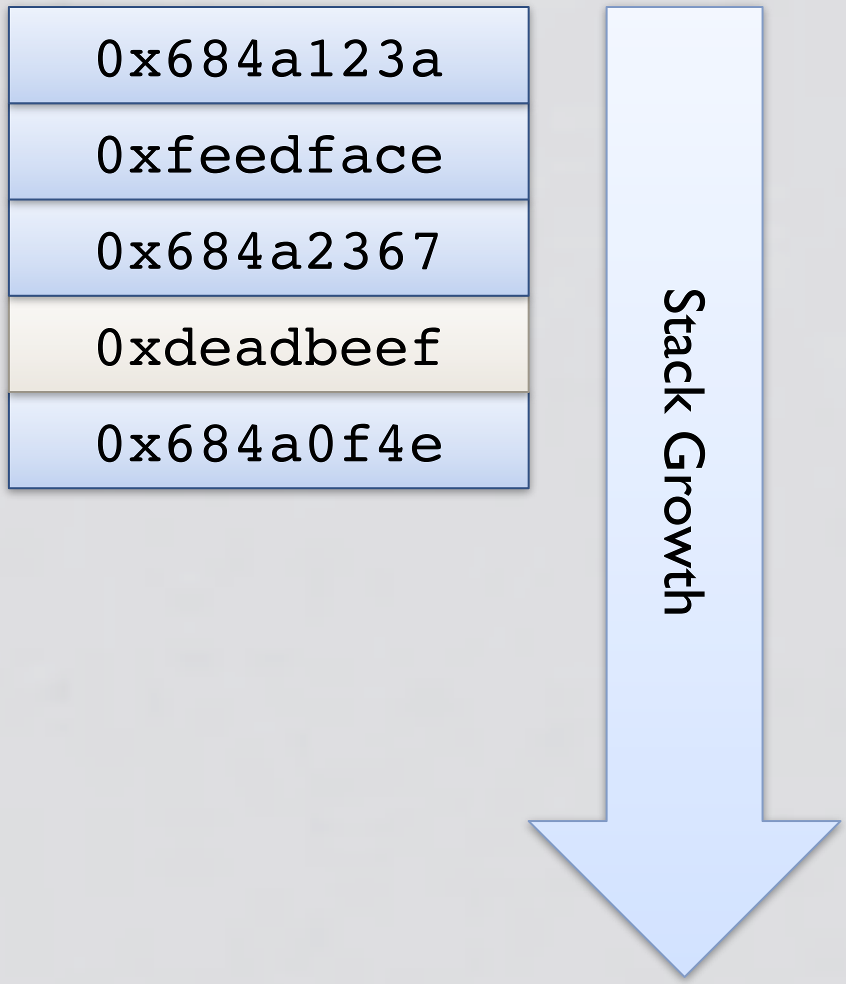
Return-Oriented Write4 Gadget

```
684a0f4e:
    pop eax
    ret
684a2367:
    pop ecx
    ret
684a123a:
    mov [ecx], eax
    ret
```



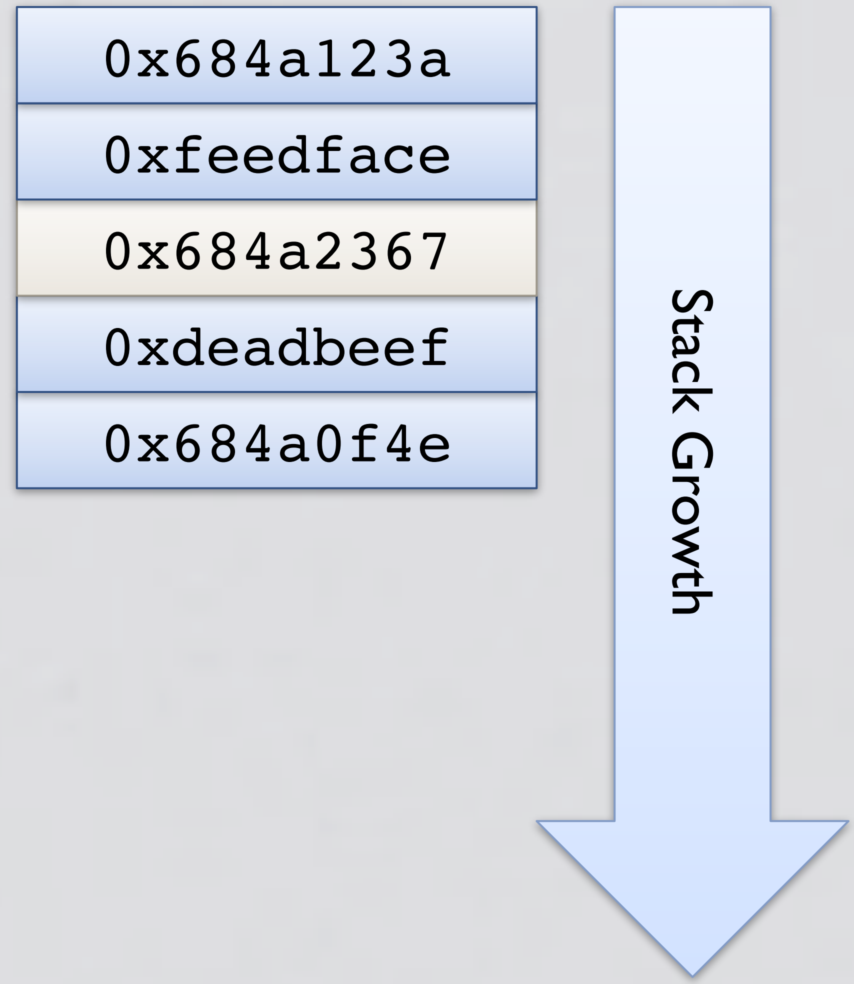
Return-Oriented Write4 Gadget

```
684a0f4e:
    pop eax
    ret
684a2367:
    pop ecx
    ret
684a123a:
    mov [ecx], eax
    ret
```



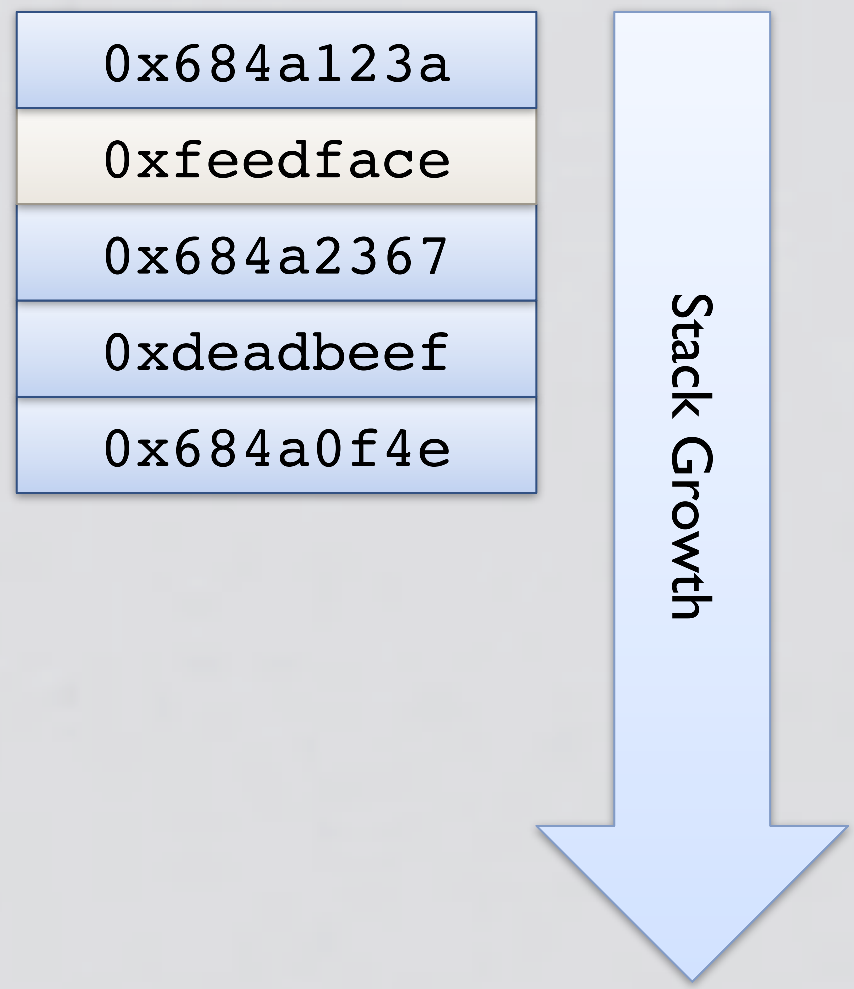
Return-Oriented Write4 Gadget

```
684a0f4e:
    pop eax
    ret
684a2367:
    pop ecx
    ret
684a123a:
    mov [ecx], eax
    ret
```



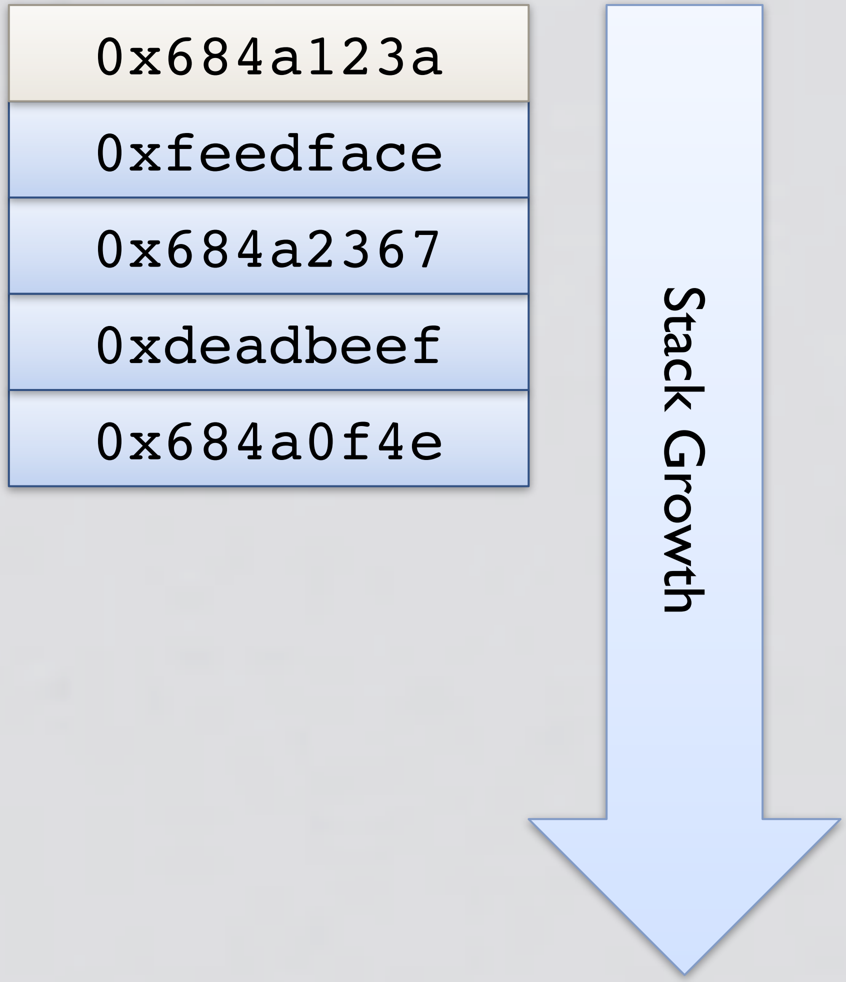
Return-Oriented Write4 Gadget

```
684a0f4e:
    pop eax
    ret
684a2367:
    pop ecx
    ret
684a123a:
    mov [ecx], eax
    ret
```



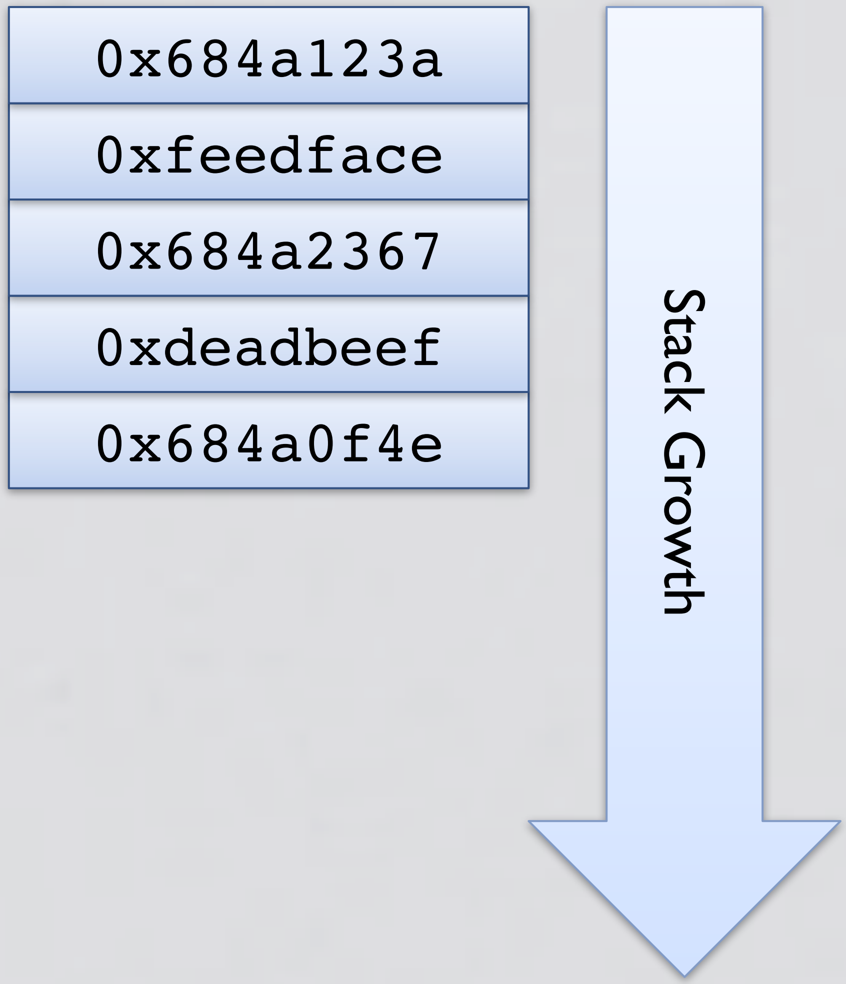
Return-Oriented Write4 Gadget

```
684a0f4e:
    pop eax
    ret
684a2367:
    pop ecx
    ret
684a123a:
    mov [ecx], eax
    ret
```



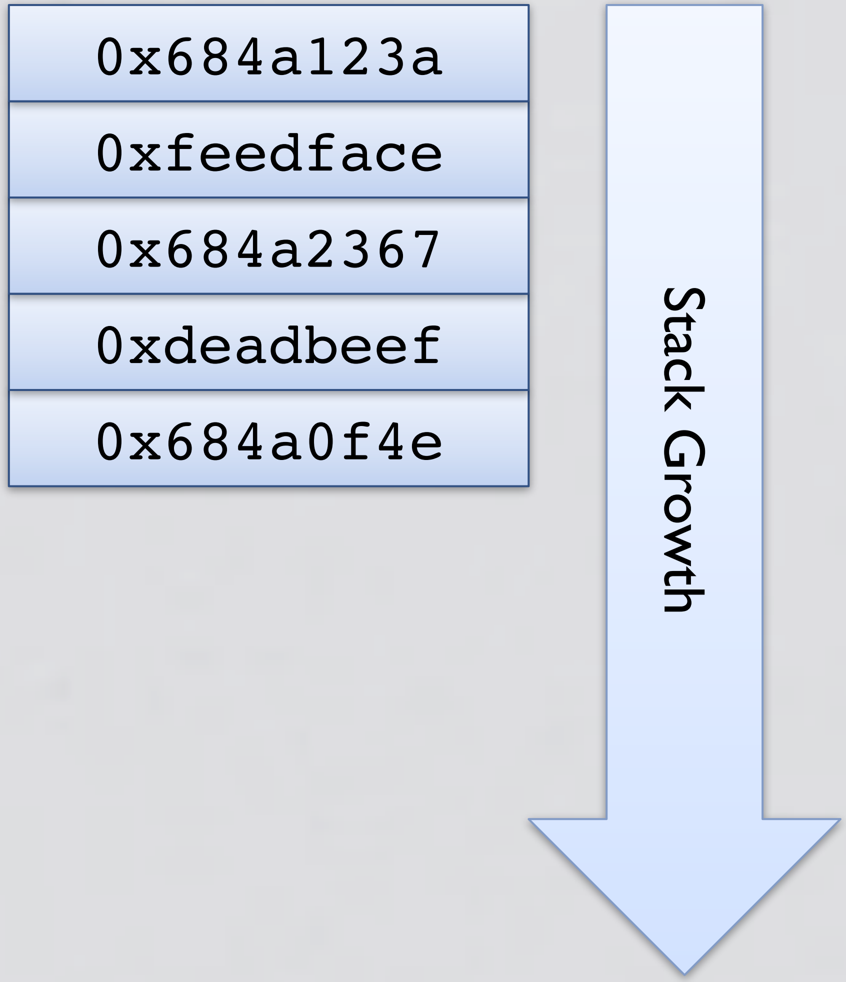
Return-Oriented Write4 Gadget

```
684a0f4e:
    pop eax
    ret
684a2367:
    pop ecx
    ret
684a123a:
    mov [ecx], eax
    ret
```



Return-Oriented Write4 Gadget

```
684a0f4e:
    pop eax
    ret
684a2367:
    pop ecx
    ret
684a123a:
    mov [ecx], eax
    ret
```



Address Space Layout Randomization

- Almost all exploits require hard-coding memory addresses
- If those addresses are impossible to predict, those exploits would not be possible
- ASLR moves around code (executable and libraries), data (stacks, heaps, and other memory regions)
- Windows Vista randomizes DLLs at boot-time, everything else at run-time

Bypassing ASLR

- Poor entropy
  - Sometimes the randomization isn’t random enough or the attacker may try as many times as needed
- Memory address disclosure
  - Some vulnerabilities or other tricks can be used to reveal memory addresses in the target process
  - One address may be enough to build your exploit

IE7 .NET User Control ASLR Bypass

- Internet Explorer allowed .NET user controls to be loaded into the IE process
- .NET assemblies are PE executables and DLLs
- The loader would honor the preferred load address of the DLL
- DLL can specify permissions of memory segments
- We can load chosen data at a chosen location with chosen memory permissions (RWX)