Runtime Attacks: Buffer Overflow and Return-Oriented Programming

Prof. Dr.-Ing. Ahmad-Reza Sadeghi
M.Sc. Lucas Davi

Course Secure, Trusted and Trustworthy Computing, Part 1
System Security Lab
http://trust.cased.de
Technische Universität Darmstadt

January 14, 2011

TECHNISCHE UNIVERSITÄT DARMSTADT

System Security Lab

Motivation: Runtime Attacks

- **Runtime attacks are major threats to today's applications**
  - Control flow of an application is compromised at runtime
  - Typically, runtime attacks include **injection of malicious code**
- **Reasons for runtime attacks**
  - Software is written in unsafe languages such as C/C++
    $\Rightarrow$ Thus, it suffers from **various memory-related vulnerabilities**
- **Most prominent example: Buffer overflow**

Motivation: Buffer Overflow

- **Are known for 2 decades**
- **Various techniques exist**
  - Stack Smashing
  - Heap Overflow
  - Integer Overflow
  - Format String

Countermeasures

- $W \oplus X$ – **Writable Xor Executable**
  - Prevents execution of injected code by marking memory pages either writable or executable
  - Implemented in Linux [PaXa] and Windows DEP (Data Execution Prevention) [Mic06]
  - Supported by chip manufactures such as Intel and AMD (NX/XD Bit)
- **ASLR – Address Space Layout Randomization**
  - Randomizes base addresses of memory segments
  - Realized in Linux PaX Kernel Patch [PaXb]
  - Enabled for Windows Vista and Windows 7 [HT07]
- **Compiler Extensions**
  - Mitigate buffer overflows by introducing stack canaries, pointer encryption, bound checkers, variable reordering, etc.

Despite many countermeasures buffer overflows are still major threats of today's applications

Buffer Overflow Vulnerabilities: Some Statistics

- **Still a major threat (e.g., in Internet Explorer or Acrobat Reader, etc.)**

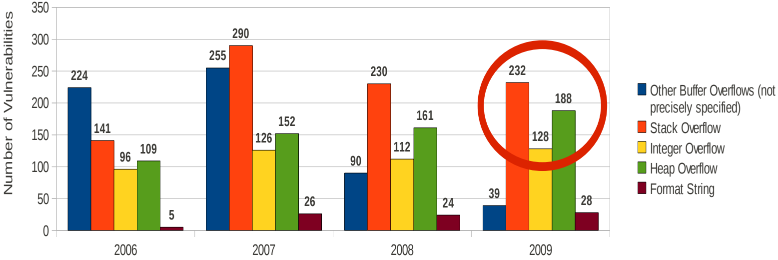


Figure: Buffer Overflows according to NIST Vulnerability Database

- **First observations**
  - Many applications are still suffering from buffer overflow vulnerabilities that allow code injection
  - Modern systems enforce $W \oplus X$ to prevent code injection attacks
- **On the other hand new attack techniques bypass** $W \oplus X$

Return-Oriented Programming

Return-Oriented Programming

Arbitrary (Turing-complete) computation without the need to

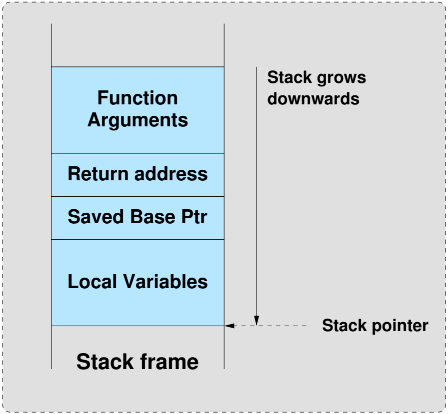
- inject malicious code
- call any library function
- modify the original code

Background and General Idea

- **Target of Buffer Overflow Attacks**
  - Subvert the usual execution flow of a program by redirecting it to a **injected (malicious) code**
- **The attack consists of**
  1. **injecting new (malicious) code** into some writable memory area,
  2. and **changing a code pointer** (usually the return address) in such a way that it points to the injected malicious code.
- **Code Injection**
  - Code can be injected by overflowing a local buffer allocated on the stack
  - The target of the injected code is usually to launch a shell to the adversary
  - Therefore the injected code is often referred to as **shellcode**

The Stack Frame

- **To understand how a buffer overflow attack works, we take a deeper look at the stack frame and its elements**



TECHNISCHE UNIVERSITÄT DARMSTADT

System Security Lab

The Stack Frame (cntd.)

- **Stack is a last in, first out (LIFO) memory area whereas the Stack Pointer (SP) points to the top word on the stack**
- **On the x86 architecture the stack grows downwards**
- **The stack can be accessed by two basic operations**
  1. **Push** elements onto the stack (SP is **decremented**)
  2. **Pop** elements off the stack (SP is **incremented**)
- **Stack is divided into individual stack frames**
  - Each function call (**call** instruction) sets up a new stack frame on top of the stack

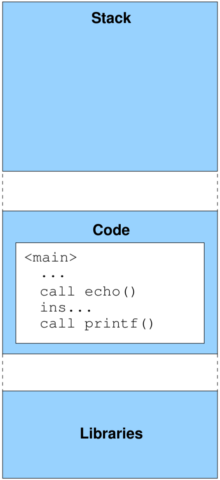
  1. **Function arguments**
  2. **Return address**
     - Upon function return (i.e., a **ret** instruction is issued), control transfers to the code pointed to by the return address (i.e., control transfers back to the caller of the function)
  3. **Saved Base Pointer**
     - Base pointer of the calling function
     - Variables/arguments are accessed via an offset to the base pointer
  4. **Local variables**

Vulnerable program

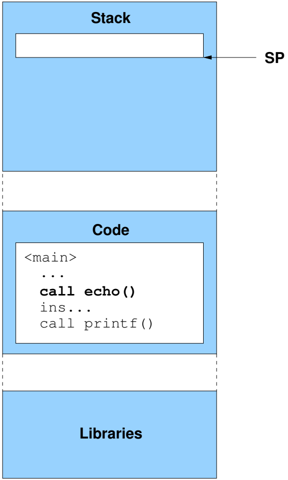
- **Simple Echo program suffering from a stack overflow vulnerability**
- **The gets() function does not provide bounds checking**

```c
#include <stdio.h>
void echo()
{
    char buffer[80];
    gets(buffer);
    puts(buffer);
}
int main ()
{
    echo();
    printf("Done");
    return 0;
}
```

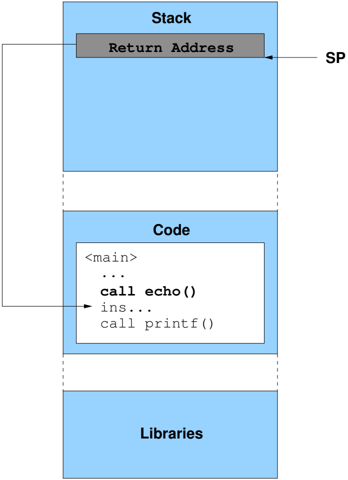
(1) Program starts



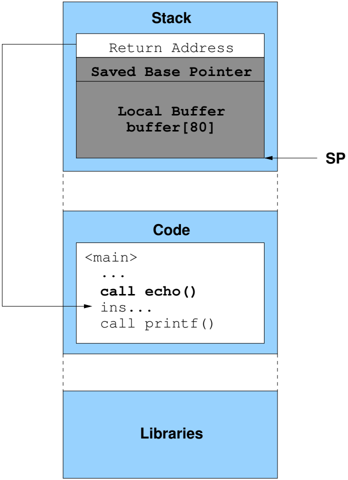
(2) The echo() function is called



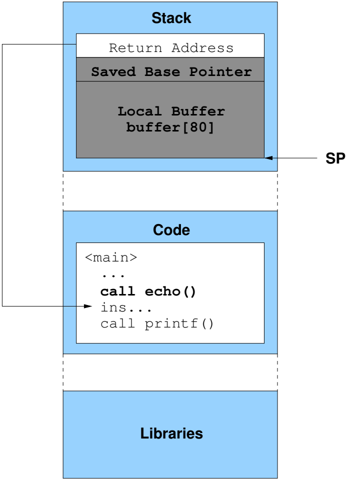
(3) Call instruction pushes return address onto the stack



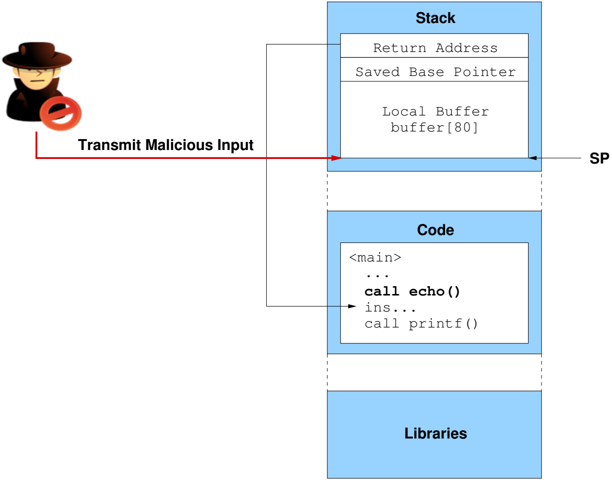
(4) Allocation of saved base pointer and buffer



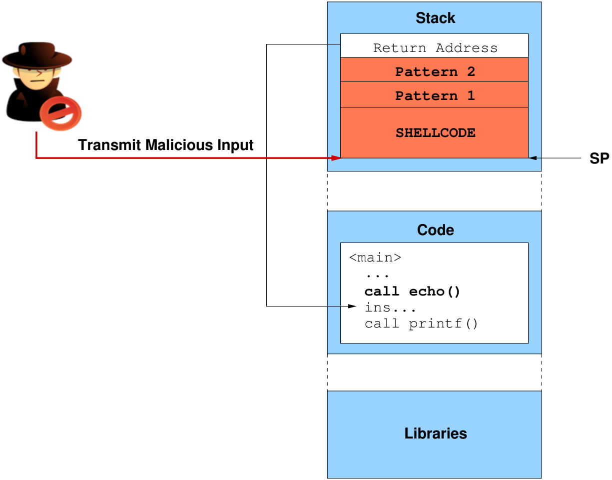
(5) echo() calls gets(buffer) function



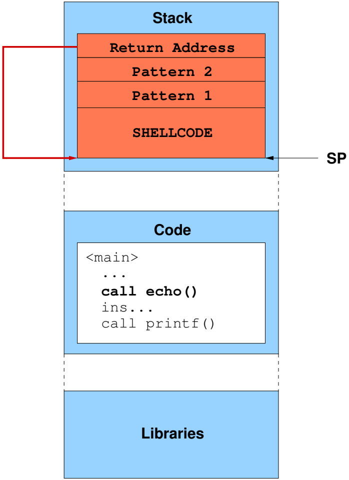
(6) Adversary transmits malicious code



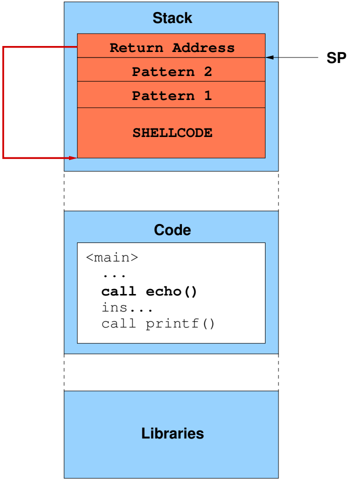
(7) Malicious code contains shellcode, pattern bytes, . . .



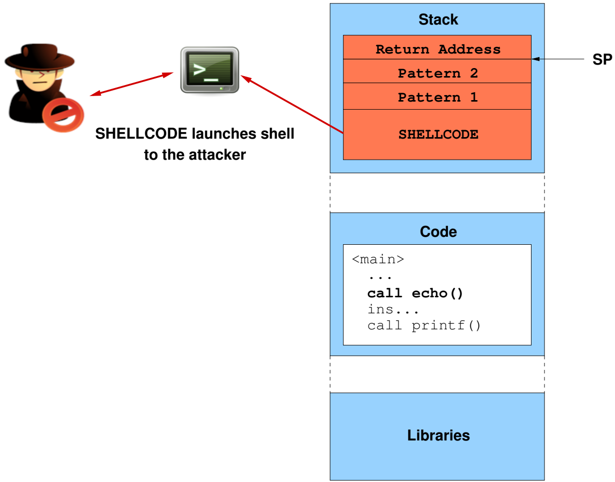
(8) . . . , and a new return address



(9) Before echo() returns to main, SP is updated



(10) echo() issues return resulting in execution of shellcode



Conclusion and Limitations

- **Why the attack is possible?**
  - The *gets()* function provides **no bounds-checking**
  - C/C++ includes various functions providing **no bounds-checking**, e.g.,
    - *strcpy()*: Copies a string into a buffer
    - *strcat()*: Concatenates two strings
    - *scanf()*: Read data from stdin (Standard Input)
- **General defense against code injection attacks is $W \oplus X$**
  - With $W \oplus X$ memory pages can be either marked writable or executable
  - Stack is marked writable
  - Hence, the adversary can only inject his malicious code, but cannot execute it

Return-into-Libc Attacks

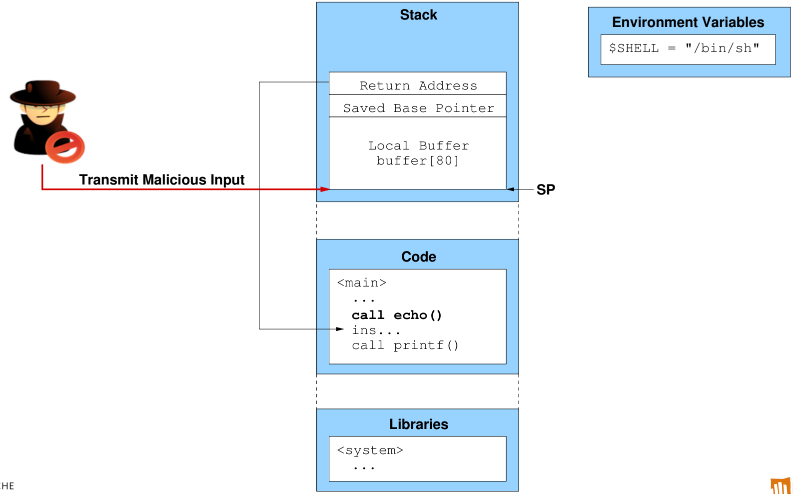
- **Basic idea of return-into-libc**
  - **Instead of injecting code use existing code**
  - Subvert the usual execution flow by redirecting it to **functions** in linked system libraries
  - The process's image consists of
    1. **writable** memory areas like stack and heap,
    2. and **executable** memory areas such as the code segment and the linked system libraries
  - The target for useful code can be found in the C library **libc**
- **The C library libc**
  - Libc is linked to nearly every Unix program
  - This library defines system calls and other basic facilities such as *open()*, *malloc()*, *printf()*, *system()*, *execve()*, etc.
    - E.g., **system ("/bin/sh")**
- **The corresponding attack is referred to as return-into-libc attack**

Useful Functions in Libc

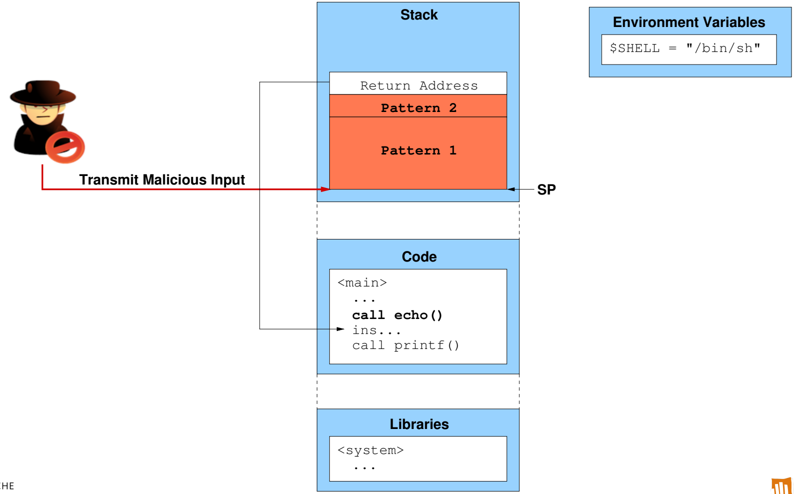
- **Libc provides the following useful functions to the adversary**
  - The **system()** function
    - Executes a new program within a running program.
  - Example: **system ("/bin/sh")**
    - This function executes the **/bin/sh** file (i.e., a new shell is launched)
  - The **execve()** function
    - Execute a new program and replace the (old) running program.
  - Example: **execve (argv[0], argv, NULL);**
    - **argv** is a string array, whereas **argv[0] = "/bin/sh"**
    - This function launches a new shell and replaces the running program

Attack Example

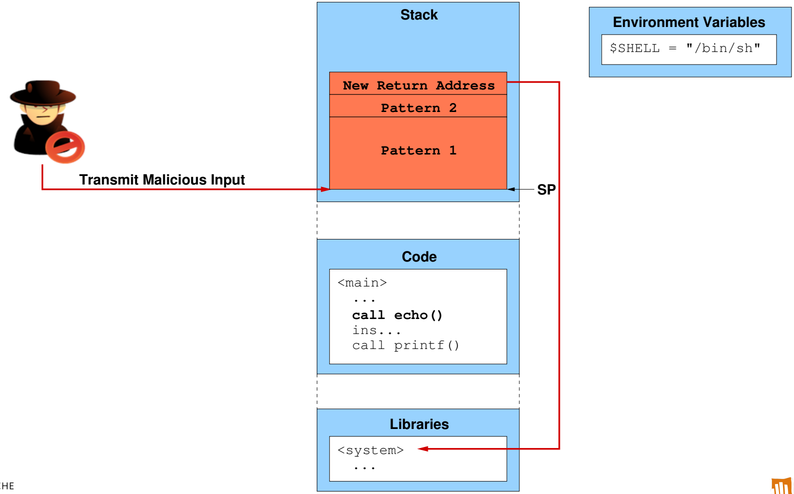
(1) Adversary transmits malicious input



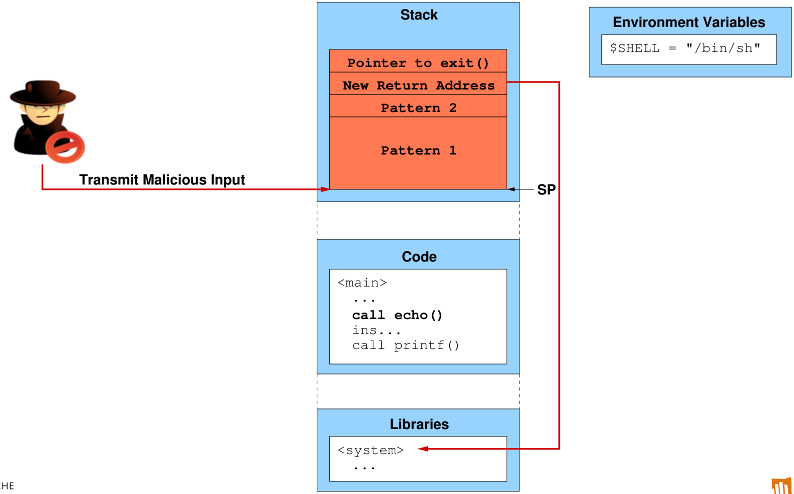
(2) Input contains pattern bytes, . . .



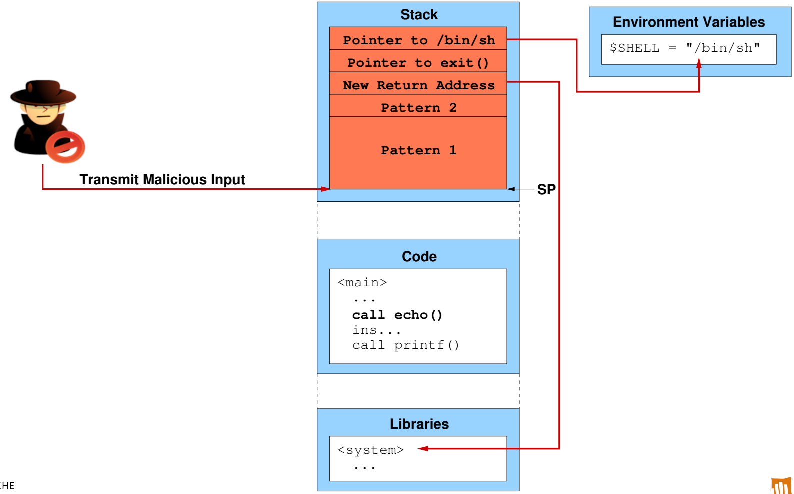
(3) . . . , a new return address pointing to system(), . . .



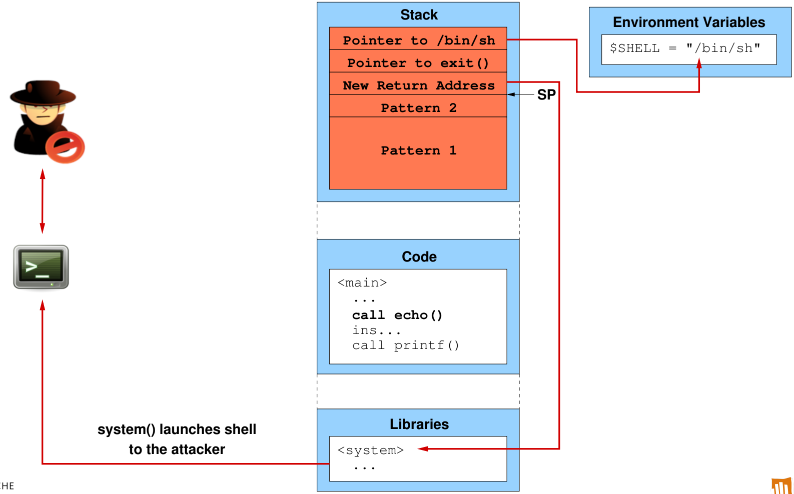
(4) . . . , a return address for system(), . . .



(5) . . . , and a pointer to the /bin/sh string



(6) When echo() returns, system() launches a new shell



Limitations

- **Return-into-libc attacks bypass security mechanisms such as the $W \oplus X$ model, but suffer from the following restrictions**
  1. The adversary relies on functions available in libc ⇒ The designers of libc could eliminate functions such as **system()**.
  2. The adversary can only invoke one function after the other ⇒ No branching is possible

The Big Picture

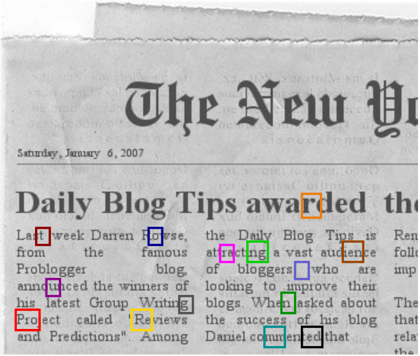

The New Yo

Saturday, January 6, 2007

Daily Blog Tips awarded the

Last week Darren Rowse, from the famous Problogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that

The Big Picture

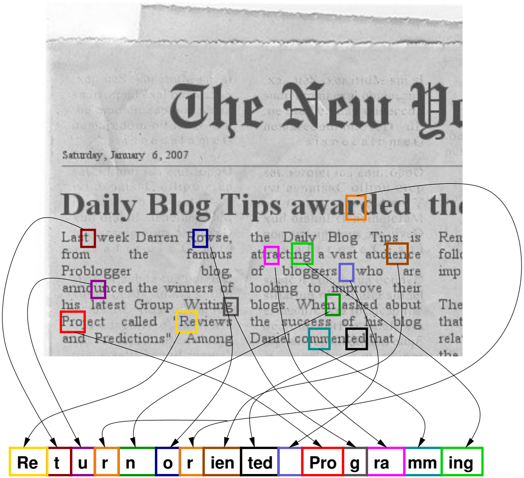

The New

Saturday, January 6, 2007

Daily Blog Tips awarded the

Last week Darren Rowse, from the famous Problogger blog, announced the winners of his latest Group Writing Project called "Reviews and Predictions". Among the Daily Blog Tips is attracting a vast audience of bloggers who are looking to improve their blogs. When asked about the success of his blog Daniel commented that

The Big Picture



TECHNISCHE UNIVERSITÄT DARMSTADT

System Security Lab

Architectures

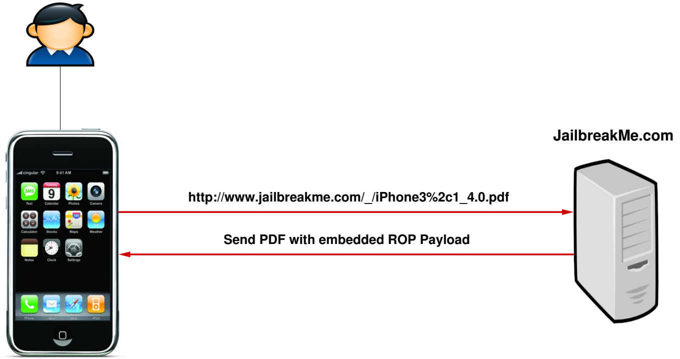
- **ROP attacks are applicable on a broad range of architectures**
  1. Intel x86 [Sha07]
  2. The SPARC Machine [BRSS08]
  3. Atmel AVR [FC08]
  4. Z80 Voting Machines [CFK^{+}09]
  5. PowerPC [Lin09]
  6. ARM [Kor09]

Real-World Exploits

- **Apple iPhone**
  - JailbreakMe [Hal10]
  - Steal SMS Database [IW10]
- **Desktop PCs**
  - Acrobat Reader [jdu10]
  - Adobe Flashplayer [Ado10]
- **Special-purpose machines**
  - Z80 voting machine [CFK+09]

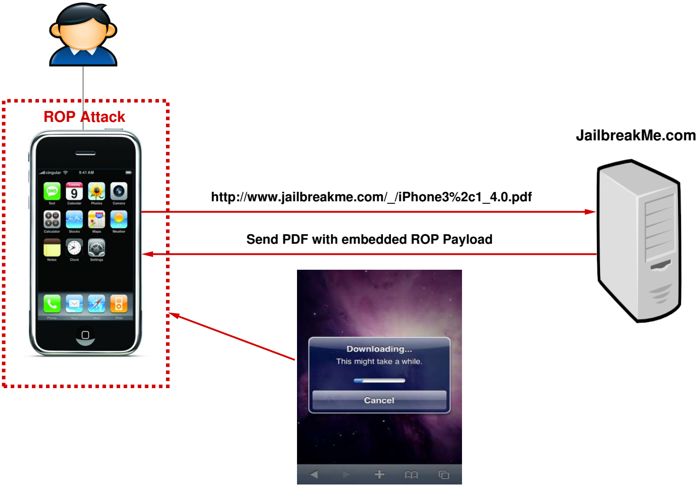
Jailbreak on Apple iPhone

(1) Download special crafted PDF file

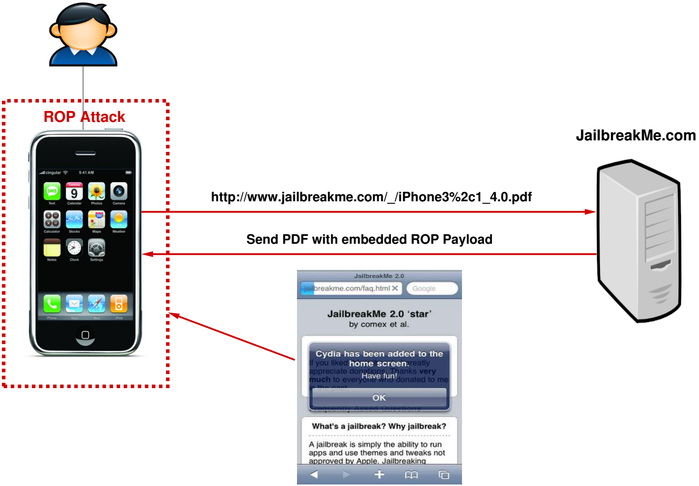


(2) ROP attack is launched

(3) Download new system files



(4) Jailbreak completed



Stealing Votes with ROP

- **Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage [CFK+09]** http://www.youtube.com/watch?v=lsfG3KPrD1I

ROP Attack on Adobe Reader

- $W \oplus X$: **Data Execution Prevention (DEP)**
  - Adobe Reader enables DEP by default
- **CVE-2010-0188**
  - Integer Overflow Vulnerability in the libtiff library of Adobe Reader
  - Use a malicious TIFF image (embedded in a PDF file) to exploit the vulnerability
  - However, Adobe Reader enables DEP by default
- **Attack**
  1. Create a malicious PDF file containing (1) ROP code and (2) arbitrary shellcode
  2. When the user opens the file, the malicious PDF first exploits the integer vulnerability
  3. Afterwards, ROP is used to exploit $W \oplus X$ to allocate a memory page marked as writable (W) and executable (X)
  4. Finally the shellcode is copied to that memory page (by means of ROP) and executed.

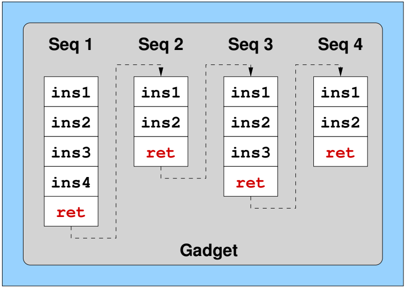
How does ROP actually work?

General Idea of ROP

- **Idea**
  - Perform **arbitrary computation** with return-into-libc techniques
- **Approach**
  - Use small **instruction sequences** (e.g., of libc) instead of using whole functions
  - Instruction sequences range from 2 to 5 instructions
  - All sequences end with a **ret** instruction
  - Instruction sequences are chained together to a **gadget**
  - A gadget performs a particular task (e.g., load, store, xor, or branch)
  - Afterwards, the adversary enforces his desired actions by **combining the gadgets**

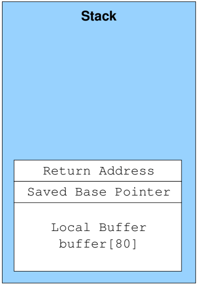
Relation of Instruction Sequences and Gadgets

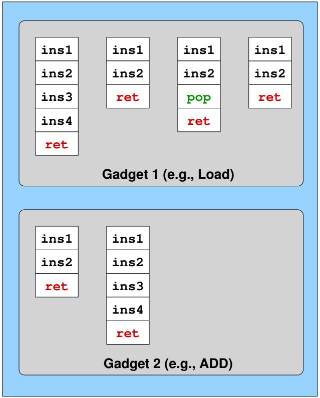
- **Instruction sequence**
  - A sequence of instructions ending in a ret instruction (return)
- **Gadget**
  - Consists of several instruction sequences



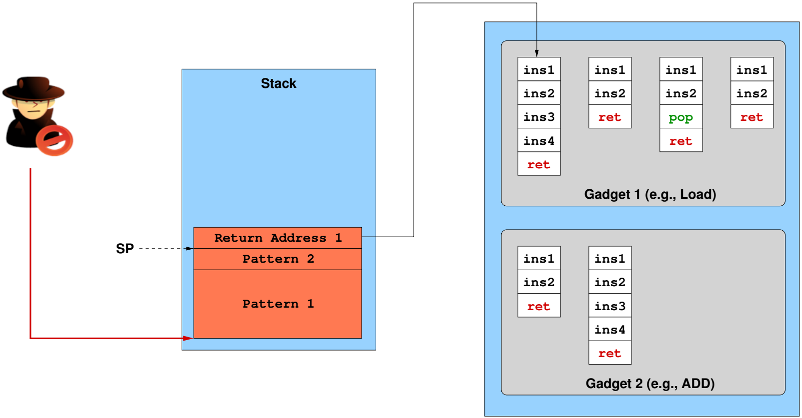
Attack Example

(1) Program is waiting for input from the user

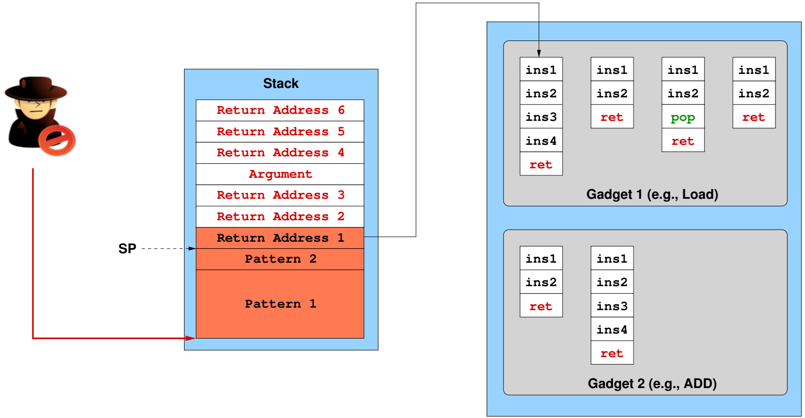




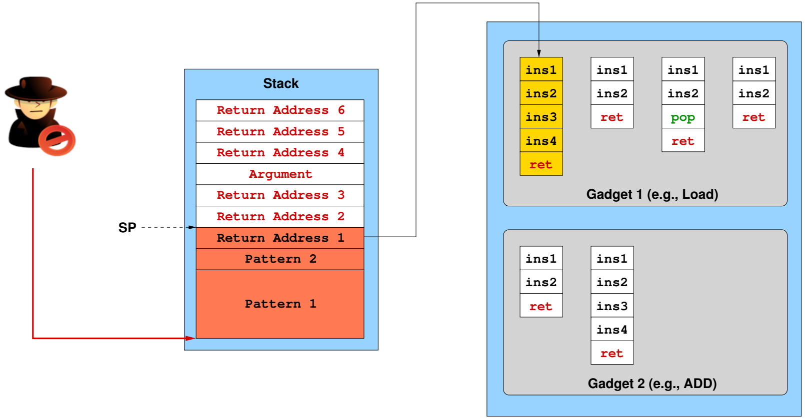
(2) Adversary overflows the buffer



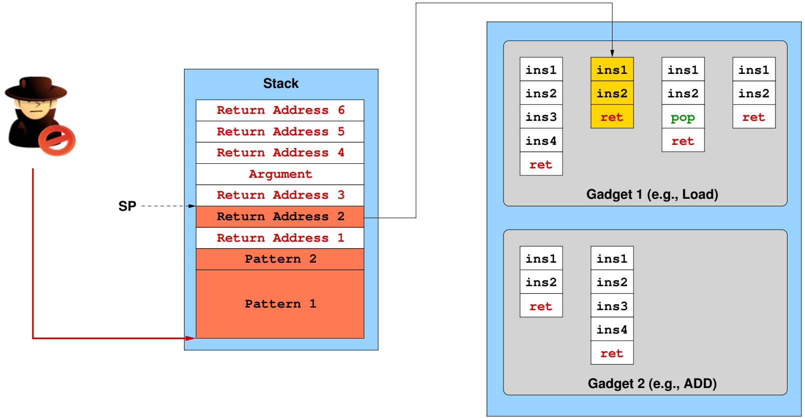
(3) Input contains return addresses and one argument



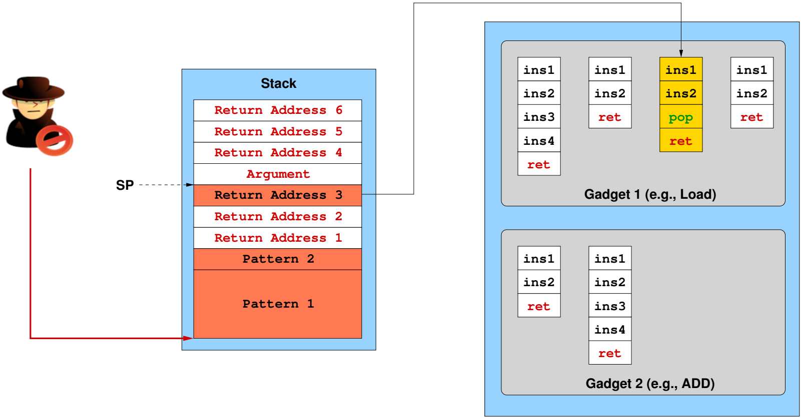
(4) foo() returns and first sequence is executed



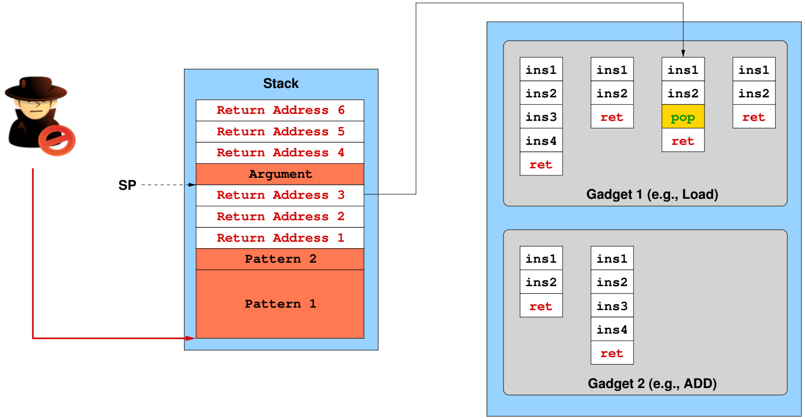
(5) Return instruction transfers control to next sequence



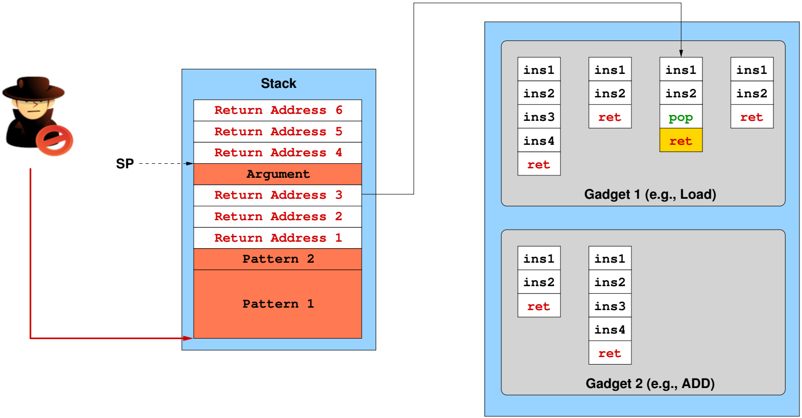
(6) Return of Sequence 2 transfers control to Sequence 3



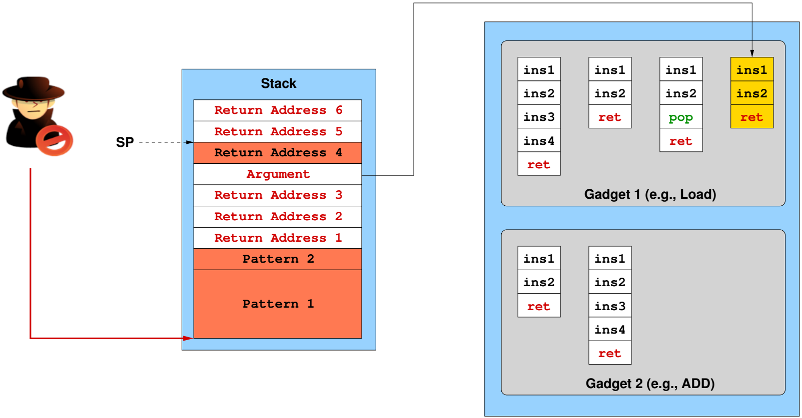
(7) Pop Argument off the stack



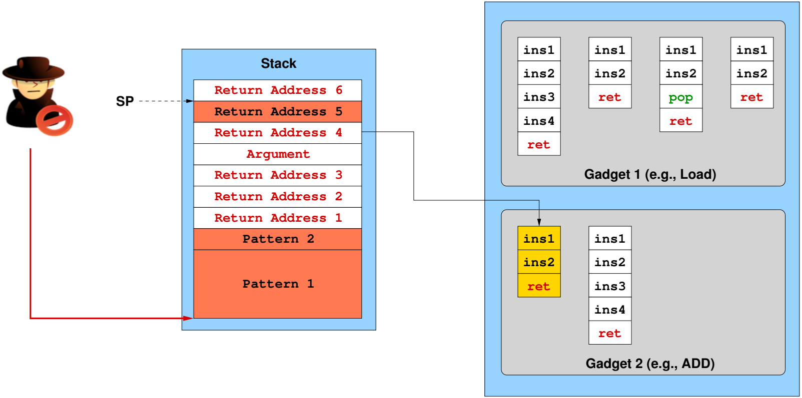
(8) Return instruction of Sequence 3 has been reached



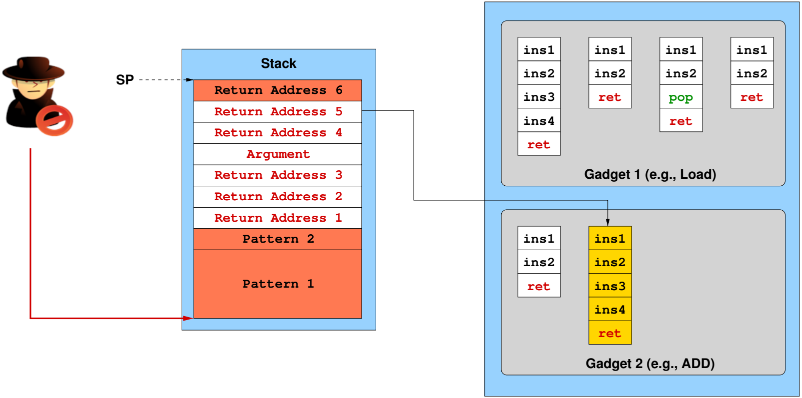
(9) Return of Sequence 3 transfers control to Sequence 4



(10) Return of Sequence 4 transfers control to Gadget 2



(11) Return of Sequence 1 transfers control to Sequence 2



Unintended Instruction Sequences

- **Unintended instruction sequences**
  - A sequence of instructions ending in a **ret** instruction that was **never intended** by the programmer
  - These sequences can be found by **jumping in the middle of a valid instruction** resulting in a new unintended instruction sequence
- **Unintended instruction sequences can be found for the x86 architecture for two reasons**
  - **Variable-length instructions**: Instructions are not of fixed size
  - **Unaligned memory access**: If the native machine word is of size N then an unaligned memory access means reading from an address that is not divisible by N.

Find Unintended Instruction Sequences

- **Consider the following instructions contained in libc**

Byte values	Assembler	Comment
`b8 13 00 00 00`	`mov $0x13,%eax`	`/* move 0x13 to the %eax register */`
`e9 c3 f8 ff ff`	`jmp 3aae9`	`/* jump to (relative) address 3aae9 */`

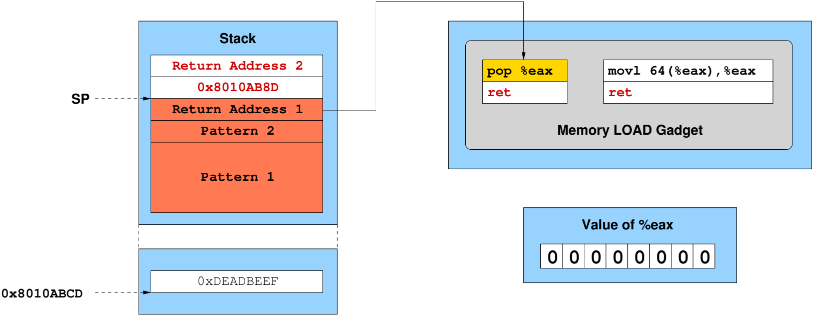
- **Instead of starting the interpretation of the byte stream at b8, starting at the third byte 00 results in following unintended instruction sequence**

Byte values	Assembler	Comment
`00 00`	`add %al,(%eax)`	`/* add register value of %al to the word */` `/* pointed to by the %eax register */`
`00 e9`	`add %ch,%cl`	`/* add registers %cl and %ch */`
`c3`	`ret`	`/* return instruction */`

Gadget Example: Memory Load

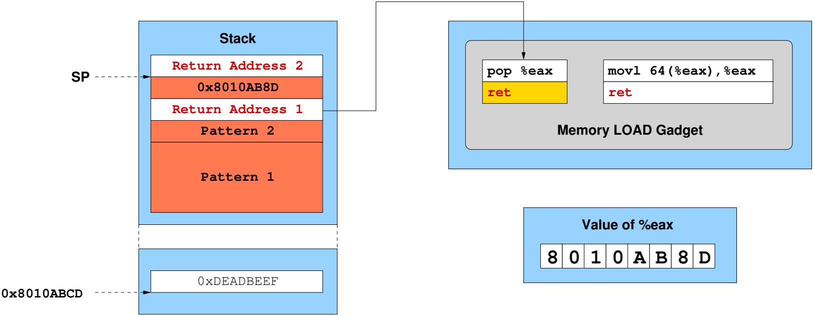
(1) Sequence 1 starts execution

- **Goal:** Load the word 0xDEADBEEF (pointed to by 0x8010ABCD) into the %eax register



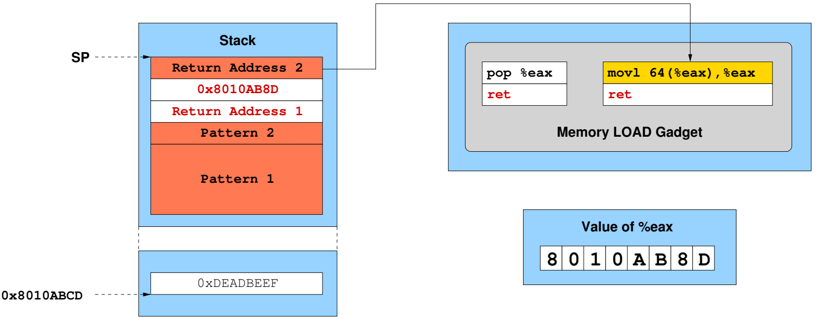
(2) Pop 0x8010AB8D in register %eax

- **Goal:** Load the word 0xDEADBEEF (pointed to by 0x8010ABCD) into the %eax register



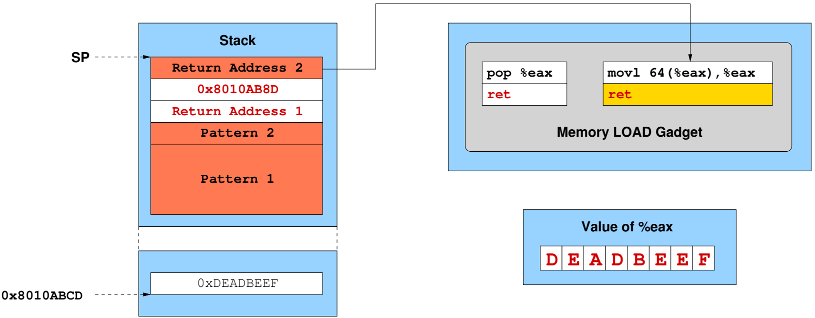
(3) Return instruction transfers control to Sequence 2

- **Goal:** Load the word 0xDEADBEEF (pointed to by 0x8010ABCD) into the %eax register



(4) Move 0xDEADBEEF in register %eax

- **Goal:** Load the word 0xDEADBEEF (pointed to by 0x8010ABCD) into the %eax register

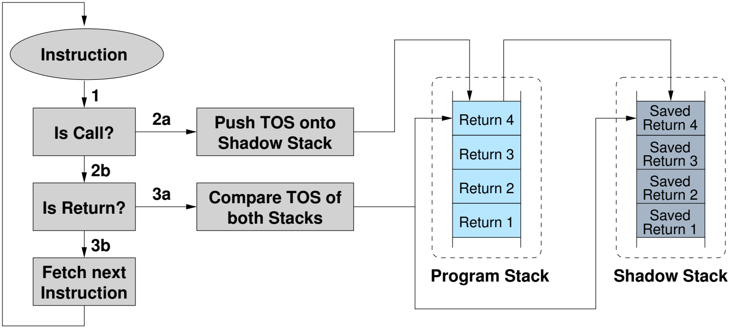


How to protect return addresses from malicious modification?

Compiler Based Solutions

- **Selected Approaches**
  - Place a **canary** before the return address
  - Backup return addresses onto a separate **shadow stack**
- **Realizations**
  1. Examples for canary based solutions
     - StackGuard [CPM+98]
     - ProPolice [Hir]
  2. Examples for shadow stack based solutions
     - Return Address Defender [CH01]
     - Stack Shield [Ven]
- **Limitations and disadvantages**
  - Compiler solutions require access to source code
  - Recompilation
  - In general, not able to detect unintended instruction sequences

Shadow Stack Approach



Hardware Facilitated Solutions

- **Approach**
  - Use existing hardware features or new hardware modules to enforce return address protection
- **Realizations**
  - Embedded microprocessor [FPC09]
    - Split the stack into data-only and call/return addresses-only parts
    - Enforce access control on call/return stack
  - StackGhost [FS01]
    - Stack Cookies XORed against return addresses
    - Solution specific to SPARC
- **Limitation**
  - Require new hardware features [FPC09] or are based on unique features of a special system [FS01]

Dynamic Binary Instrumentation based on a JIT-Compiler

- **Approach**
  - Add **instrumentation code** by compiling an instruction block to new instructions at runtime (**JIT – Just In Time Compilation**)
  - JIT-based instrumentation allows the detection of unintended sequences
- **Realizations**
  - Program Shepherding [KBA02]
    - Checks if a return targets a valid call site, i.e., a return has to target an instruction which is preceded by a call instruction
  - ROPdefender [DSW10]
    - Checks each return address against valid return addresses hold in a separate shadow stack
  - Measure return frequency: DynIMA [DSW09], DROP [CXS+09]
- **Limitations**
  - JIT-based instrumentation adds high performance overhead
  - Solutions based on measuring the frequency of returns can be bypassed by executing longer sequences

Is it possible to bypass return address checkers?

Return-Oriented Programming without Returns [CDD+10]

ROP without Returns

- **Results**
  - Countermeasures that protect return addresses are bypassed
  - Attack technique for **Intel x86** and **ARM**
  - **Turing-complete gadget set** and practical attack instantiation for both platforms **without** any return instruction
- **Approach**
  - Use **return-like sequences**
  - Candidates are **indirect jumps**
    - On Intel: `jmp *%eax`
    - On ARM: `blx r3`
- **Obstacles**
  - Target register (`%eax, r3`) must be initialized before
  - Returns automatically update the stack pointer; indirect jumps not

Return-Like Sequences

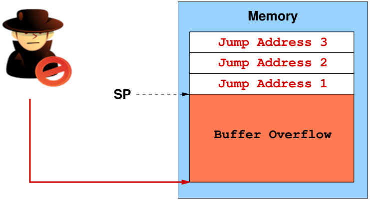
- **On Intel**
  - `pop %eax; jmp *%eax`
    1. Pop target address into `%eax`
    2. The `pop` instruction automatically increases the stack pointer by four bytes (similar to a return)
    3. Jump to the address stored in `%eax`
- **On ARM**
  - No pop-jump sequence present
  - Use **Update-Load-Branch** Sequence
    1. (Update) – `adds r6,#4`: Add four bytes to `r6`
    2. (Load) – `ldr r5, [r6]`: Load target address into `r5`
    3. (Branch) – `blx r5`: Branch to target address
- **Problem**
  - Return-like sequences for both platforms are rare

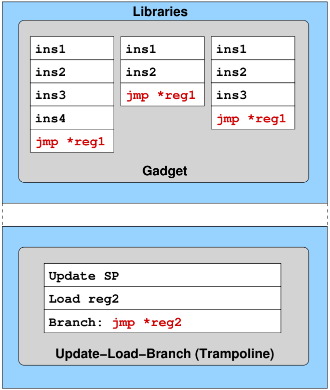
Trampoline

- **Solution**
  - Use a unique Update-Load-Branch (ULB) sequence after each instruction sequence
  - ULB is used as a **trampoline**
  - All other sequences have to end in an indirect jump to ULB

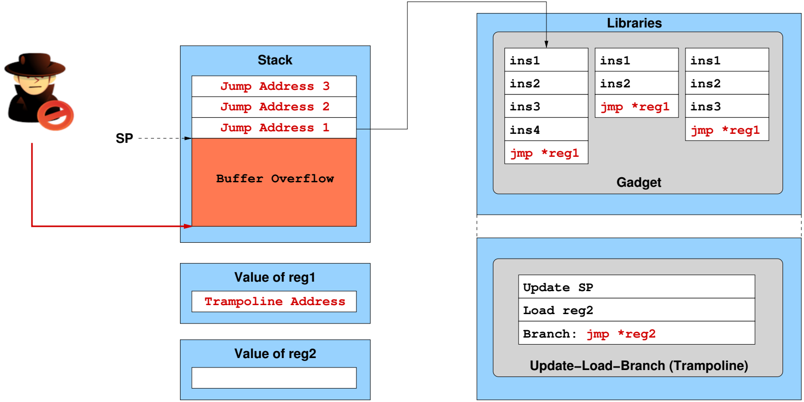
Attack Example

(1) Adversary launches a buffer overflow

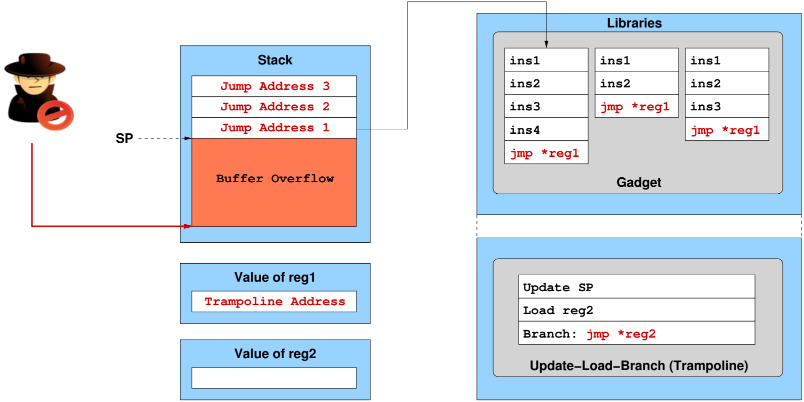




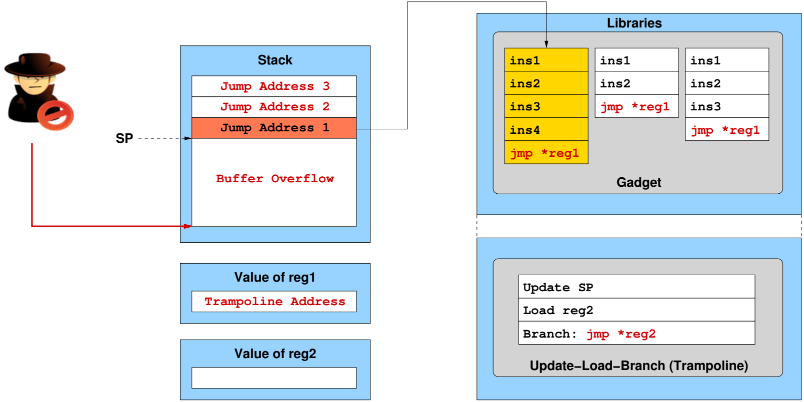
(2a) reg1 is initialized with the address of the trampoline



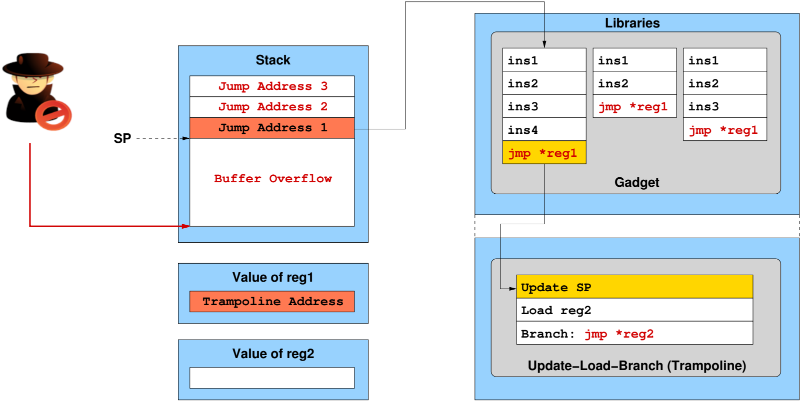
(2b) Jump Address 1 points to Sequence 1



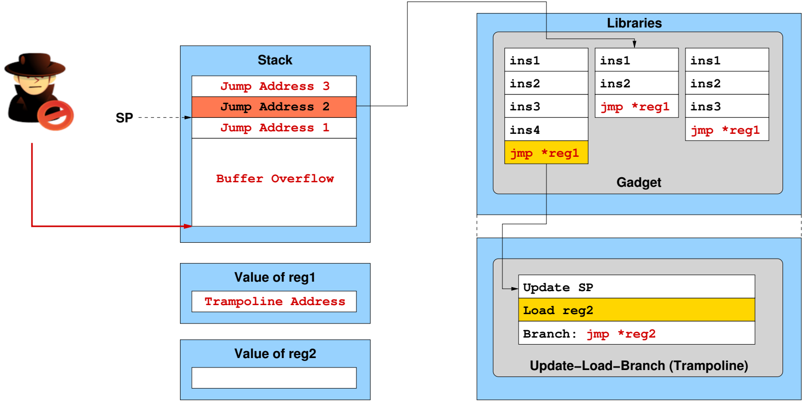
(3) Sequence 1 is executed



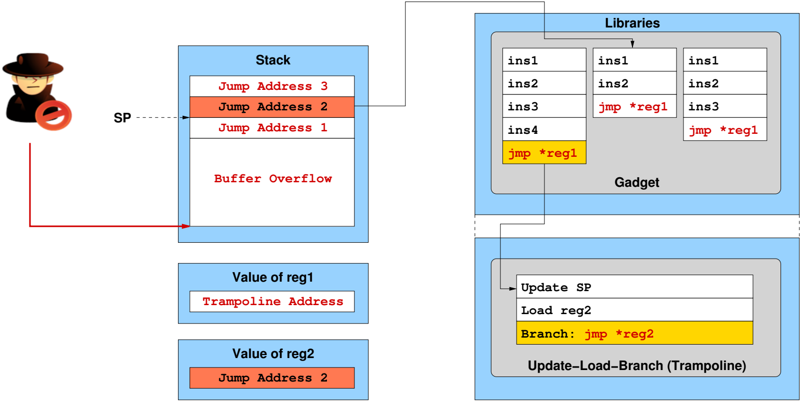
(4) Jump to Trampoline enforced



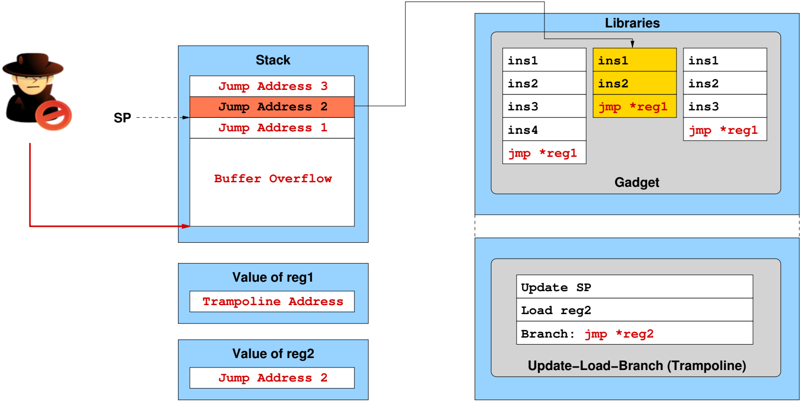
(5) Stack pointer is updated



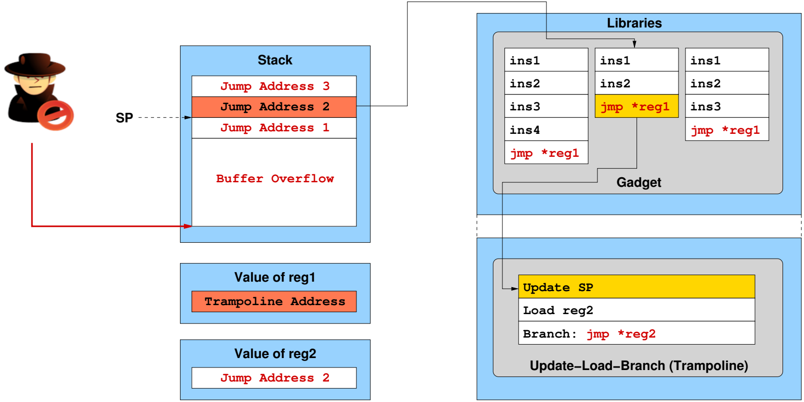
(6) Jump Address 2 is loaded in register reg2



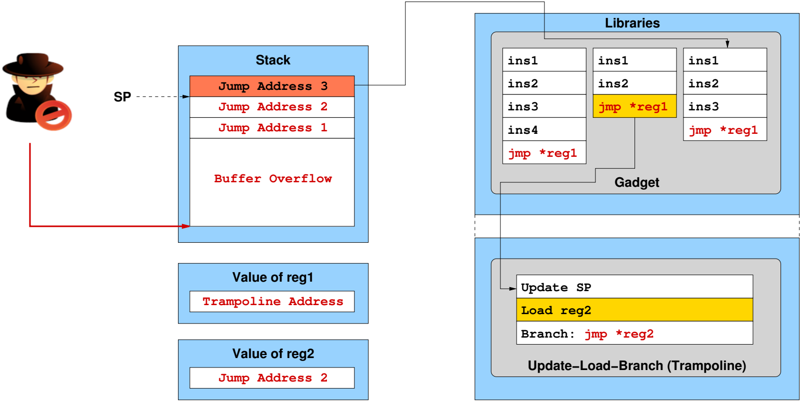
(7) Branch to Sequence 2 is enforced



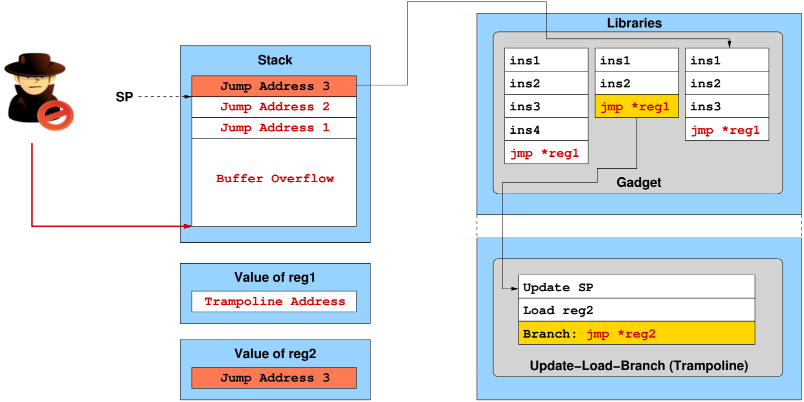
(8) Jump to Trampoline is enforced



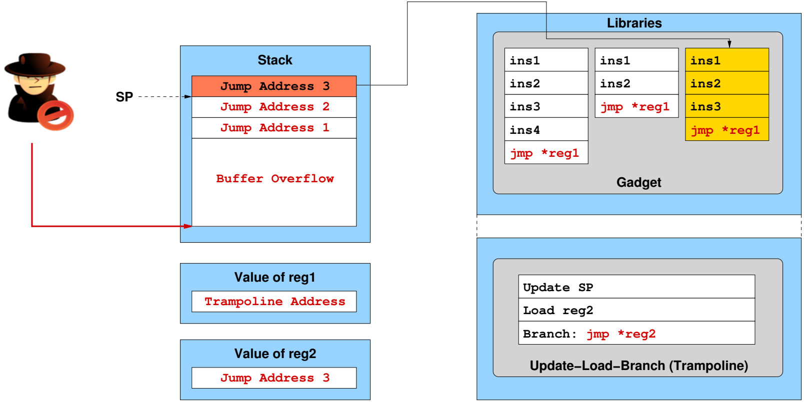
(9) Stack Pointer is updated



(10) Jump Address 3 is loaded in register reg2



(11) Branch to Sequence 3 is enforced



Attack instantiation

- **Start the ROP attack**
  - Goal: Get control of the stack pointer and the instruction pointer
    - Usually stack smashing is used for conventional ROP
    - However, we want to avoid the use of any return instruction
  - Several techniques are described in [CDD+10]
- **Example: Setjmp Buffer Overwrite**
  - *setjmp()*/*longjmp()* are system calls to allow non-local gotos
    1. *setjmp()*: Store current stack frame and processor registers in a special buffer (the setjmp buffer)
    2. *longjmp()*: Return to saved stack frame and reset processor registers to the values stored in the setjmp buffer
  - Setjmp Buffer Overwrite
    - A buffer is allocated before the setjmp buffer
    - Overflow the buffer with ROP payload and overwrite contents of the setjmp buffer
    - When *longjmp()* is called the ROP code is executed

Countermeasures

- **Control Flow Integrity (CFI) [ABEL05, ABE+06]**
  - Derives a control flow graph from a given binary
  - Labels all branch targets with a special instruction (a label ID)
  - Rewrites the binary to include new instructions that check at runtime if an indirect branch (return, jump, call) targets a valid label ID
- **Limitations of CFI**
  - Requires debugging information stored in Windows PDB files
  - CFI is built on top of the dynamic binary instrumentation framework Vulcan which is not publicly available

Address Space Layout Randomization (ASLR)

- **Approach**
  - **Randomizes the base address** of each segment (stack, heap, libraries, etc.)
    $\Rightarrow$ Thus, an attacker does not know the start addresses of instruction sequences
- **Realizations**
  - Linux PaX Kernel Patch [PaXb]
  - Available for Windows since MS Vista [HT07]
- **Limitations**
  - Parts of the code are not randomized, allowing an attacker to construct some gadgets
    - [RMPB09]: Overwrite GOT (Global Offset Table) entries with new values.
  - Information leakage and brute-force attacks possible
    - E.g., see [SjGM+04, SD08]

G-Free: Gadget-Less Binaries

- **G-Free [OBL+10]: Technique and Approach**
  - Compiler-based approach to defeat ROP through gadget-less binaries
  - Requires recompilation
  - Possible unintended instruction sequences are eliminated through code transformations
  - Protection of intended return instructions
    - Return addresses are encrypted against a random cookie
  - Protection of intended jump and call instructions
    - Upon function entry, a function-unique cookie (function identifier xor random key) is stored on the stack
    - All indirect jumps/calls are extended with a validation block
    - The indirect jump/call is only allowed if the validation block successfully decrypts the cookie

References I

[ABE+06] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, George C. Necula, and Michael Vrable. XFI: software guards for system address spaces. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[ABEL05] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity: Principles, implementations, and applications. In *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

[Ado10] Adobe Systems. Security Advisory for Flash Player, Adobe Reader and Acrobat: CVE-2010-1297. `http://www.adobe.com/support/security/advisories/apsa10-01.html`, 2010.

[BRSS08] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 27–38. ACM, 2008.

[CDD+10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *CCS '10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 559–572. ACM, 2010.

References II

[CFK+09] Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In *Proceedings of EVT/WOTE 2009*, 2009.

[CH01] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems*, pages 409–417. IEEE Computer Society, 2001.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, pages 63–78. USENIX Association, 1998.

[CXS+09] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting return-oriented programming malicious code. In Atul Prakash and Indranil Gupta, editors, *Fifth International Conference on Information Systems Security (ICISS 2010)*, volume 5905 of *Lecture Notes in Computer Science*, pages 163–177. Springer, 2009.

[DSW09] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *Proceedings of the 4th ACM Workshop on Scalable Trusted Computing (STC'09)*, pages 49–54. ACM, 2009.

References III

[DSW10] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. `http://www.trust.rub.de/media/trust/veroeffentlichungen/2010/03/20/ROPdefender.pdf`, March 2010.

[FC08] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 15–26. ACM, 2008.

[FPC09] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the 1st Workshop on Secure Execution of Untrusted Code (SecuCode'09)*, pages 19–26. ACM, 2009.

[FS01] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 55–66. USENIX Association, 2001.

[Hal10] Josh Halliday. Jailbreakme released for apple devices. `http://www.guardian.co.uk/technology/blog/2010/aug/02/jailbreakme-released-apple-devices-legal`, August 2010.

[Hir] Hiroaki Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp`.

[HT07] Michael Howard and Matt Thomlinson. Windows vista isv security. `http://msdn.microsoft.com/en-us/library/bb430720.aspx`, April 2007.

References IV

[IW10] Vincenzo Iozzo and Ralf-Philipp Weinmann. Ralf-Philipp Weinmann & Vincenzo Iozzo own the iPhone at PWN2OWN. `http://blog.zynamics.com/2010/03/24/ralf-philipp-weinmann-vincenzo-iozzo-own- the-iphone-at-pwn2own/`, Mar 2010.

[jdu10] jduck. The latest adobe exploit and session upgrading. `http://blog.metasploit.com/2010/03/latest-adobe-exploit-and-session.html`, 2010.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206. USENIX Association, 2002.

[Kor09] Tim Kornau. Return oriented programming for the ARM architecture. `http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf`, 2009. Master thesis, Ruhr-University Bochum, Germany.

[Lin09] Felix Lindner. Developments in Cisco IOS forensics. CONFidence 2.0. `http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf`, November 2009.

[Mic06] Microsoft. Data Execution Prevention (DEP). `http://support.microsoft.com/kb/875352/EN-US/`, 2006.

[OBL+10] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *ACSAC'10, Annual Computer Security Applications Conference*, December 2010.

References V

[PaXa] PaX Team. `http://pax.grsecurity.net/`.

[PaXb] PaX Team. PaX address space layout randomization (ASLR). `http://pax.grsecurity.net/docs/aslr.txt`.

[RMPB09] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC 2009)*. IEEE, 2009.

[SD08] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections in Windows Vista. `http://www.phreedom.org/research/bypassing-browser-memory-protections/`, August 2008. Presented at Black Hat 2008.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS '07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 552–561. ACM, 2007.

[SjGM+04] Hovav Shacham, Eu jin Goh, Nagendra Modadugu, Ben Pfaff, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 298–307. ACM, 2004.

[Ven] Vendicator. Stack Shield: A "stack smashing" technique protection tool for Linux. `http://www.angelfire.com/sk/stackshield`.