For a successful memory exploits, it usually involves:

- Put malicious code at a predictable location in memory, usually masquerading as data
- Trick vulnerable program into passing control to it
- Overwrite saved EIP, such as return address, function pointers, GOT tables, etc, with the values pointing to attackers malicious payload
A Basic Exploit

**Return Oriented Programming (ROP)**

- **Heap Spray**

**Summary**

### What Does W⨁X Not Prevent?

- Can still corrupt stack... (there is no buffer overflow prevention)
  - ... or function pointers or critical data on the heap
- As long as "saved EIP" points into existing code, W⨁X protection will not block control transfer

### Return-into-libc exploits

- Overwrite saved EIP with the address of any library routine, arrange memory to look like arguments

Does not look like a huge threat

- Attacker cannot execute arbitrary code
- ... especially if `system()` is not available

---

**DEP**

- Marks regions of memory as writable or executable, not both
- Prevents attacker from writing a payload to memory, then directly executing it
- Example: Microsoft’s DEP (Data Execution Prevention)
- This blocks (almost) all code injection exploits

### Hardware support

- AMD “NX” bit, Intel “XD” bit (in post-2004 CPUs)
- Makes memory page non-executable

### Widely deployed

- Windows (since XP SP2), Linux (via PaX patches), OpenBSD, OS X (since 10.5)
Overwritten saved EIP need not point to the beginning of a library routine.

Any existing instruction in the code image is fine.
  - Will execute the sequence starting from this instruction.
  - What if instruction sequence contains RET?

Execution will be transferred... to where?
- Read the word pointed to by stack pointer (ESP).
- Guess what? Its value is under attacker’s control! (why?)

Use it as the new value for EIP.
- Now control is transferred to an address of attacker’s choice!
- Increment ESP to point to the next word on the stack.

Can chain together sequences ending in RET.
- Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique” (2005)
- What is this good for?

Answer [Shacham]: everything.
- Return-oriented Programming
- Turing-complete language
- Build “gadgets” for load-store, arithmetic, logic, control flow, system calls
- Attack can perform arbitrary computation using no injected code at all!

The fact that the "ret" instruction will pass control flow to an arbitrary memory address pointed to by ESP.

Small snippets of code that end in a "ret" instruction - called gadgets.

Gadgets can be built up consecutively on the stack, executing small groups of instructions before returning control flow to the next gadget.

Gadgets are found in executable sections of libraries or the program itself, therefore avoiding execution directly on the stack.
Return Oriented Programming

- Instruction pointer (EIP) determines which instruction to fetch and execute
- Once processor has executed the instruction, it automatically increments EIP to next instruction
- Control flow by changing value of EIP

Ordinary Programming

- Stack pointer (ESP) determines which instruction sequence to fetch and execute
- Processor doesn’t automatically increment ESP
  - But the RET at end of each instruction sequence does

No-ops

- No-op instruction does nothing but advance EIP
- Return-oriented equivalent
  - Point to return instruction
  - Advances ESP
- Useful in a NOP sled ()
Immediate Constants

- Instructions can encode constants
- Return-oriented equivalent
  - Store on the stack
  - Pop into register to use

Control Flow

- Ordinary programming
  - (Conditionally) set EIP to new value
- Return-oriented equivalent
  - (Conditionally) set ESP to new value

Gadgets: Multi-instruction Sequences

- Sometimes more than one instruction sequence needed to encode logical unit
- Example: load from memory into register
  - Load address of source word into EAX
  - Load memory at (EAX) into EBX
Unintended Instructions

Gadget design

- Gadgets are built from found code sequences.
- Gadgets are intermediate organizational unit and perform well-defined operations, such as:
  - load-store operations
  - arithmetic & logic operations
  - control flow
  - invoking system calls
- The set of gadgets is Turing complete, so return-oriented programming could construct arbitrary computations.

Gadgets are built from found code sequences.
Gadgets are intermediate organizational unit and perform well-defined operations, such as:
- load-store operations
- arithmetic & logic operations
- control flow
- invoking system calls
The set of gadgets is Turing complete, so return-oriented programming could construct arbitrary computations.

Loading a Constant to Register

Loading From Memory
**Storing To Memory**

- Arithmetic And Logic
  - Simple add into %eax

```
movl %eax, 24(%edx)
ret
```

```
addl (%edx), %eax
push %edi
ret
```

**Return-Oriented Shellcode**

- An application of return-oriented shellcode.
- The shellcode invokes the execve system call to run a shell.

**Steps in creating ROP Shellcode**

- Setting the system call index in %eax;
- Setting the path of the program to run in %ebx.
- Setting the argument vector argv in %ecx.
- Setting the environment vector envp in %edx.

**x86 Architecture Helps**

- Register-memory (M2R, R2M) machine
  - Plentiful opportunities for accessing memory
- Register-starved
  - Multiple sequences likely to operate on same register
- Instructions are variable-length, unaligned
  - More instruction sequences exist in libc
  - Instruction types not issued by compiler may be available
- Unstructured call/ret ABI
  - Any sequence ending in a return is useful
### SPARC: the Un-x86

- Load-store RISC machine
  - Only a few special instructions access memory
- Register-rich
  - 128 registers; 32 available to any given function
- All instructions 32 bits long; alignment enforced
- No unintended instructions
- Highly structured calling convention
  - Register windows
  - Stack frames have specific format

---

### More ROP

- **Harvard architecture**: code separate from data → code injection is impossible, but ROP works fine
  - Z80 CPU – Sequoia AVC Advantage voting machines
  - Some ARM CPUs – iPhone
- No returns = no problems
  - (Lame) defense against ROP: eliminate sequences with RET and/or look for violations of LIFO call-return order
  - Use update-load-branch sequences in lieu of returns + a trampoline sequence to chain them together
  - Read “Return-oriented programming without returns” (CCS 2010)

---

### Other Issues with W⊕X / DEP

- Some applications require executable stack
  - Example: Lisp interpreters
- Some applications are not linked with /NXcompat
  - DEP disabled (e.g., popular browsers)
- JVM makes all its memory RWX – readable, writable, executable (why?)
  - Spray attack code over memory containing Java objects (how?), pass control to them
- Return into a memory mapping routine, make page containing attack code writeable
Heap-spraying Attacks

Besides W ⊕ X / DEP, we have ASLR. How to defeat ASLR?

Heap spraying is a new technique used in exploits to facilitate arbitrary code execution.
- Targeted at browsers, document viewers, etc.
- Current attacks include IE, Adobe Reader, and Flash
- Effective in any application the allows JavaScript

http://en.wikipedia.org/wiki/Heap_spraying

History

- Heap sprays have been used occasionally in exploits since at least 2001
- Widespread use in exploits for web browsers in the summer of 2005 (Internet Explorer).
- Simple enough to understand
- Quickly write reliable exploits for many types of vulnerabilities in web browsers and web browser plug-ins.

- JavaScript
- VBScript
- ActionScript

Revealing Heap Spraying (1)

In courtesy of Ratanaworabhan et al [Nozzle] for this example

Revealing Heap Spraying (2)

In courtesy of Ratanaworabhan et al [Nozzle] for this example
In courtesy of Ratanaworabhan et al [Nozzle] for this example

Heap-spraying Attacks

**How?**
1. Attacker must have existing vulnerability (i.e., overwrite a function pointer)
2. Attacker allocates many copies of malicious code as JavaScript strings
3. When attacker subverts control flow, jump is likely to land in malicious code

**Code injection is not necessary for arbitrary exploitation.**

**Defenses that distinguish “good code” from “bad code” are hard and also useless.**

**Return-oriented programming (ROP) likely possible on every architecture, not just x86.**

**ROP shell code can be automatically generated (Q [USENIX Security 2011]) from application binary code**

**There are many ways to write robust shell code besides ROP**
References

- [CCS07] H. Shacham “The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”
- [CCS08] Buchanan et al. “When good instructions go bad: Generalizing return-oriented programming to RISC”
- [ACSAC09] “Surgically Returning to Randomized Lib(c)”