Lec 8: DEP

IS561: Binary Code Analysis and Secure Software Systems

Sang Kil Cha

Memory Corruption

Two Ways to Corrupt Memory

- Buffer overflow.
- Format string.

Integer overflows can lead to a memory corruption (but not always).

Control-Flow Hijack

Memory corruption can lead to a control flow hijack. There are two things to consider:

- How to redirect the control?
  - e.g., overwriting a control data
- Where to redirect the control?
  - e.g., injected shellcode

Food for Thought

Can we execute arbitrary commands by exploiting a memory corruption bug, but without hijacking the control flow?

Memory Defense

Prevention vs. Mitigation

Prevention vs. Mitigation

- Preventing buffer overflows.
  - Buffer overflows will never happen.
- Mitigating buffer overflows.
  - Buffer overflows will happen, but will be hard to exploit them.

Can One Prevent Buffer Overflows?

Yes. Just do ***NOT*** use **C/C++**.

Memory-Safe Langugages

F#, OCaml, Haskell, Python, etc.

```
>>> x = array('l', [1, 2, 3])
>>> x[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: array index out of range
```

Unfortunately though ...

- C/C++ is still very popular.
- Legacy code is written in C/C++.

Not so easy to ***prevent*** memory corruption.

Unfortunately though ...

- C/C++ is still very popular.
- Legacy code is written in C/C++.

Not so easy to ***prevent*** memory corruption. Hence, we'd better ***mitigate*** it. 😐

DEP

Buffer Overflow Mitigation #1: DEP

Data Execution Prevention[1] = NX (No eXecute).

Stack stores data, but not code. Therefore, we make the stack memory area ***non-executable***.

[1] DEP ***prevents*** data execution, but it does not prevent buffer overflows.

DEP Has Many Names

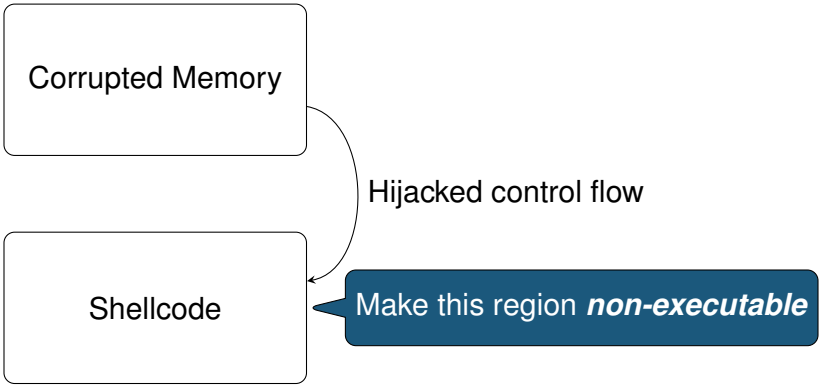
AMD Athlon™ Processor Competitive Comparison

FEATURES	AMD ATHLON™ CPU	PENTIUM® 4
Architecture Introduction	2006	2000
Infrastructure	Socket AM2	Socket LGA775
Process Technology	90 nanometer, SOI 65 nanometer, SOI	90 nanometer
64-bit Instruction Set Support	Yes, AMD64 technology	Depends, EM64T on some Pentium® 4 series
Enhanced Virus Protection for Windows® XP SP2*	Yes	Depends

W $\oplus$ X (Write XOR eXecute) Policy

- Every page should be either writable or executable, but not both.
- Even though we can put a shellcode to a writable buffer, we cannot execute it if this policy is enabled.

Defeating Control Flow Hijack with DEP



DEP on Stack

There is a special segment called "`PT_GNU_STACK`" in ELF files, which controls whether the stack should be executable or not. We use the `execstack` command to change the ELF header to control whether the binary should have an executable stack or not.

```
$ /usr/sbin/execstack -s <filename> ; clear NX flag
$ /usr/sbin/execstack -c <filename> ; set NX flag
$ /usr/sbin/execstack -q <filename> ; query NX flag
```

With NX set on the stack, return-to-stack exploits will fail.

But,

DEP does not prevent buffer overflows. It prevents return-to-stack exploits, though.

Any other ways to exploit buffer overflows?

Code-Reuse Attacks

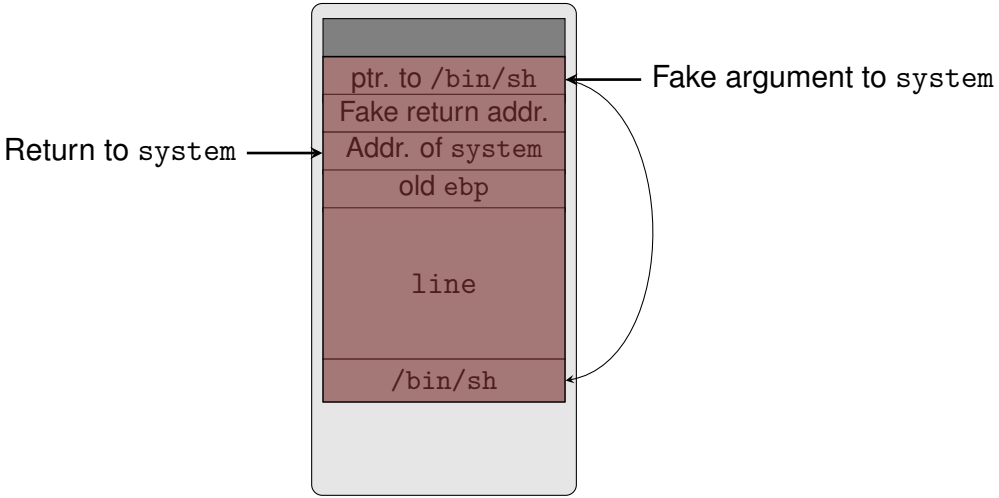
Bypassing DEP

- We can still hijack the control flow with buffer overflows.
- We can still jump to an arbitrary address of ***existing code***.

Code Reuse Attack #1: Return-to-LIBC

- LIBC is a standard library that is used by every C program.
  - e.g., `printf` is a LIBC function.
- Many useful functions in LIBC to execute.
  - `exec` family: `execl`, `execlp`, `execle`, ...
  - `system`
  - `mprotect`
  - `mmap`

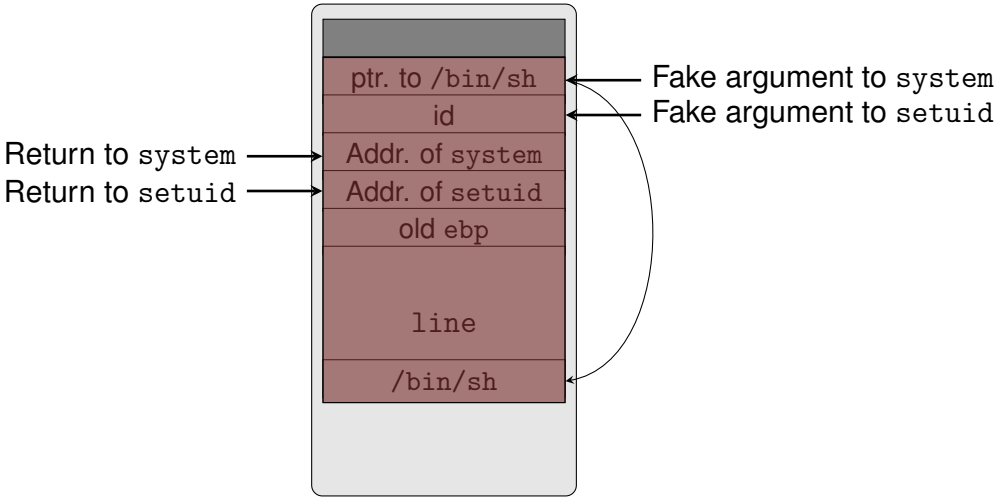
Return-to-LIBC



Question

Can we call multiple LIBC functions? For example, suppose we want to call `setuid` first, and then call `execve`.

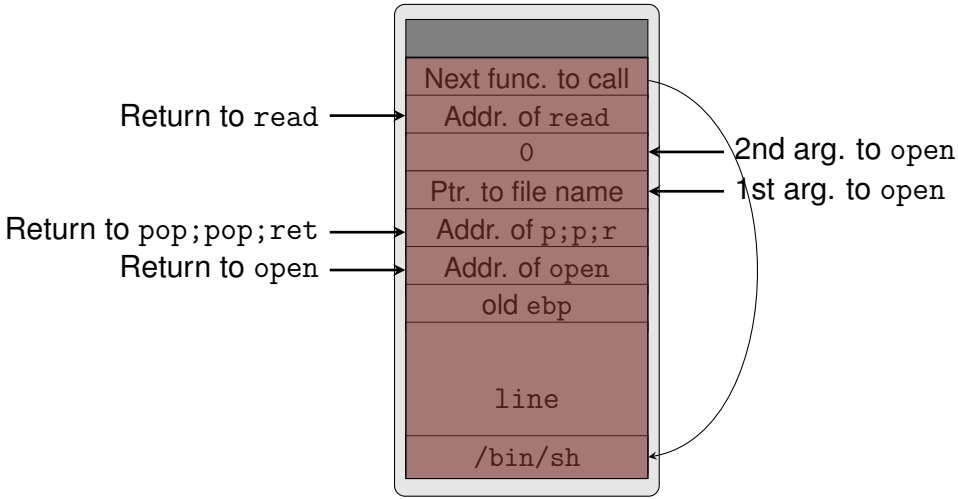
Chaining Two Function Calls



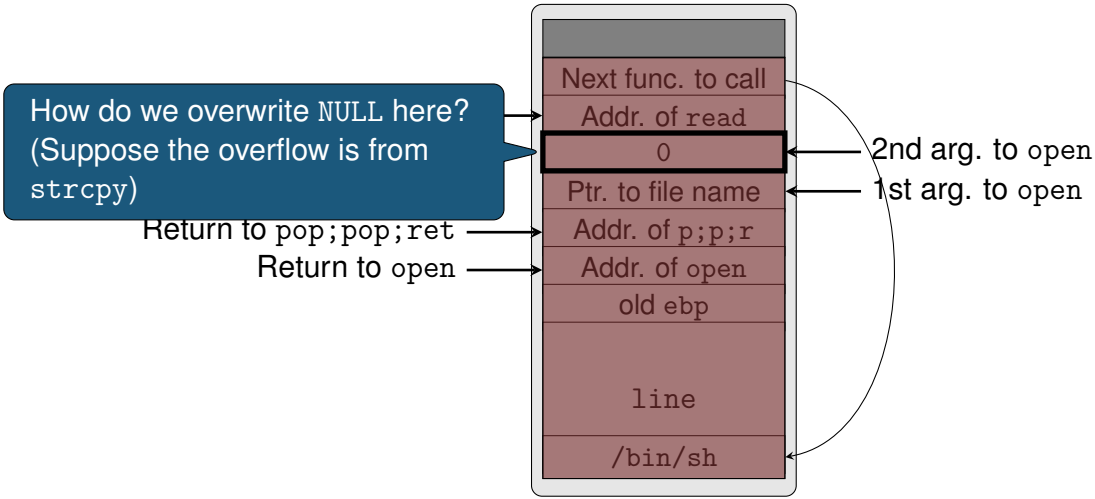
Question

Can we call multiple LIBC functions that require more than one parameter?

ESP Lifting: `open–read`



ESP Lifting: `open–read`



Generalization of Idea

The idea of jumping into a code block that ends with “ret” instruction[2] becomes the primitive of ***ROP*** (Return-Oriented Programming).

```
pop <reg>
pop <reg>
ret
```

[2] Such a code block is often referred to as a ROP ***gadget***.

ROP

Code Reuse Attack #2: ROP

Formally introduced by Hovav in ***CCS 2007***[3]

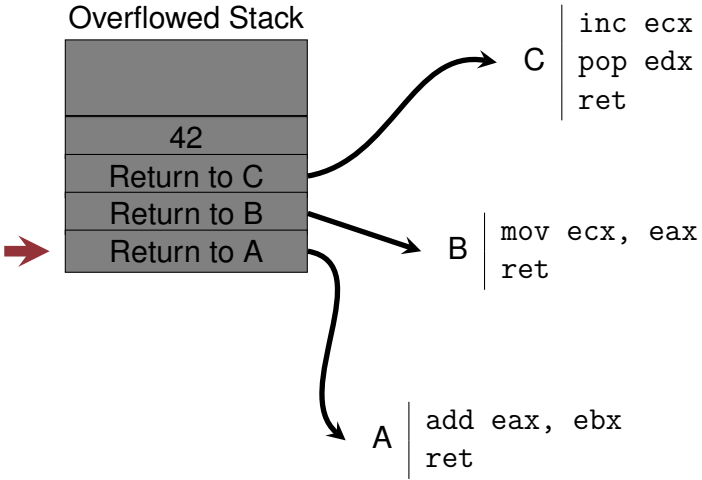
[3] The Geometry of Innocent Flesh on the Bone: Return-to-libc without Function Calls (on the x86)

Motivation of ROP

Return-to-LIBC requires LIBC function calls, but can we spawn a shell without the use of LIBC functions?

- Different versions of LIBC.
- LIBC may not be used at all.
- Some functions in LIBC can be excluded.

Return (`ret`) Chaining



Return (`ret`) Chaining

```
add eax, ebx
mov ecx, eax
inc ecx
mov edx, 42
```

C

```
inc ecx
pop edx
ret
```

B

```
mov ecx, eax
ret
```

Return chaining allows arbitrary computation!

A

```
add eax, ebx
ret
```

ROP Example

Goal: Modify `ptr` to be 0x42424242 with ROP.

```
mov [ptr], 0x42424242
```

ROP Example (cont'd)

It is very unlikely to find a gadget: `mov [ptr], 0x42424242; ret`. So, we have to connect several gadgets to construct the logic.

`pop eax` `ret`	Assign the `ptr` to `eax`
`pop ebx` `ret`	Assign 0x42424242 to `ebx`
`mov [eax], ebx` `ret`	Modify the `ptr`

Question

Can we encode a conditional branch with ROP? Suppose we want to encode the following logic:

$$\texttt{if (eax < 42) then (ecx} \leftarrow \texttt{0) else (ecx} \leftarrow \texttt{1)}$$

Conditional Jumps in ROP (3 Steps)

1. Modify (set/clear) flags of interest.
2. Transfer the flag from EFLAGS to a general-purpose register.
3. Use the flag of interest to perturb the stack pointer conditionally[4].

Several useful tricks:

- `pushf` instruction pushes EFLAGS to the stack.
- sub eax, 42 should result in `CF` = 1 when `eax < 42`.
- The result of `adc cl, cl` when `ecx = 0` is the same as `CF`.

[4] Stack pointer is a PC in ROP.

ROP Conditional Jump Example

1. `neg eax; ret`
2. `pop ecx; pop edx; ret`
3. `adc cl, cl; ret`
4. `mov [edx], ecx; ret`
5. `pop ebx; ret`
6. `neg dword [ebx+0x5e]; pop edi; pop ebp; mov esi, esi; ret`
7. `pop esi; ret`
8. `pop ecx; pop ebx; ret`
9. `and [ecx], esi; rol byte ptr [ebx+0x5e5b6cc4],0x5d; ret`
10. `add esp, [ecx]; add byte ptr [eax],al; add byte ptr[eax], cl; ret`

ROP Workflow

1. Disassemble binary.
2. Identify useful gadgets.
   - e.g., an instruction sequence that ends with `ret` is useful.
   - e.g., an instruction sequence that ends with `jmp reg` is also useful.
3. Assemble gadgets to perform some computation.
   - e.g., spawning a shell

★ Challenge: Gathering as many gadgets as possible

Many Gadgets in Regular Binaries?

x86 instructions have their lengths ranging from 1 byte to 18 bytes, i.e., it uses ***variable-length encoding***.

Therefore, there can be both intended and unintended gadgets in x86 binaries.

Disassembling x86

```
8d 4c 24 04 83 e4 f0
```

```
lea ecx, [esp+0x4]
and esp, 0xfffffff0
```

What if we disassemble the code from the second byte (`4c`)?

Disassembling x86

```
8d 4c 24 04 83 e4 f0
```

```
dec esp
and al, 0x4
and esp, 0xfffffff0
```

Totally different, but still ***valid*** instructions!

Unintended Instructions

- One can disassemble from any address in a memory page.
- We can indeed find lots of ***unintended ROP gadgets*** using this idea.

Example: Unintended `ret` Instruction

Compiler-intended instructions:

```
e8 05 ff ff ff          call 0x8048330
81 c3 59 12 00 00       add ebx, 0x1259
```

If we disassemble the same binary starting from the second byte:

```
05 ff ff ff 81          add eax, 0x81ffffffii
c3                      ret
```

Galileo Algorithm: Finding ROP Gadgets

Algorithm Galileo:
  create a node, root, representing the `ret` instruction;
  place root in the trie;
  for pos **from** 1 to `textseg_len` **do**:
    if the byte at pos is `c3`, i.e., a `ret` instruction, **then**:
      call BuildFrom(pos, root);

Procedure BuildFrom(index pos, instruction `parent_insn`):
  for step **from** 1 **to** `max_insn_len` **do**:
    if bytes[(pos - step) ... (pos - 1)] decode as a valid instruction `insn` **then**:
      ensure `insn` is in the trie as a child of `parent_insn`;
      if `insn` isn't boring **then**:
        call BuildFrom(pos - step, insn);

Galileo Algorithm: Finding ROP Gadgets

Algorithm Galileo:
  create a node, root, representing the `ret` instruction;
  place root in the trie;
  for pos **from** 1 to `textseg_len` **do**:
    if the byte at pos is `c3`, i.e., a `ret` instruction, **then**:
      call BuildFrom(pos, root);

Procedure BuildFrom(index pos, instruction `parent_insn`):
  for step **from** 1 **to** `max_insn_len` **do**:
    if bytes[(pos - step) ... (pos - 1)] decode to a valid instruction `insn` **then**:
      ensure `insn` is in the t
      if `insn` isn't boring **the**
        call BuildFrom(pos -

1. The `insn` is a `leave` instruction.
2. The `insn` is `pop ebp`.
3. The `insn` is a unconditional jump.

Program Size May Matter

Larger code $\rightarrow$ More chance to get useful gadgets.

Schwartz et al.[5] show that 100KB was enough to successfully create exploits for 80% of the binaries in `/usr/bin`.

[5] Q: Exploit Hardening Made Easy, USENIX Security 2011.

ROP without `ret`?

Return-oriented Programming without Returns, ***CCS 2010***.

```
pop eax; jmp [eax]
```

Question

How can we mitigate code reuse attacks (ROP)?

Detecting ROP

Basic idea: If `ret` instructions are frequently used within a short amount of time, then it is likelly to be a ROP attack.

- Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, ***USENIX Security 2013***.
- ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks, ***NDSS 2014***.

Detecting ROP

Basic idea: If `ret` instructions are frequently used within a short amount of time, then it is likelly to be a ROP attack.

- Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, ***USENIX Security 2013***.
- ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks, ***NDSS 2014***.

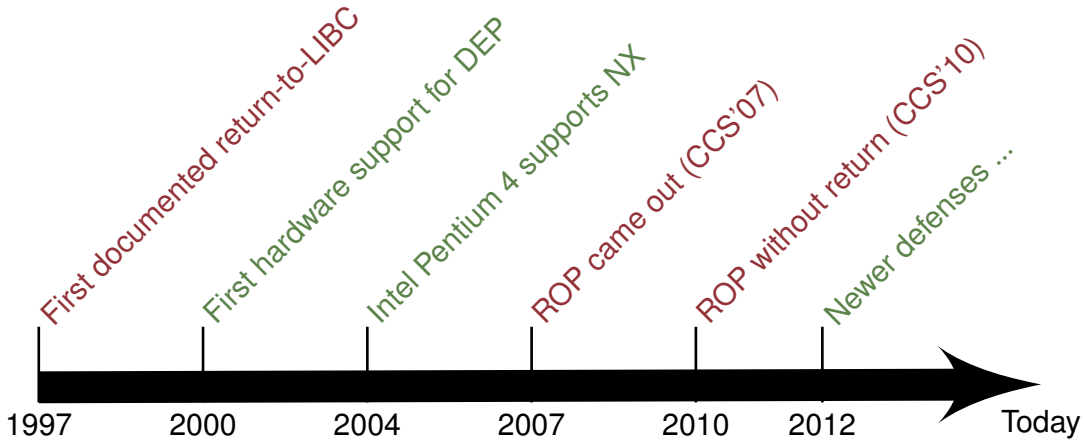
But such a simple idea suffers from both false negatives and false positives:

- Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard, ***USENIX Security 2014***.

Other Defenses?

We will discuss further throughout this course.

DEP and Code Reuse Attacks Timeline



Summary

- NX (or DEP) is one way to mitigate control flow hijacks.
- Code reuse attacks allow an attacker to bypass DEP.
- Many mitigation techniques are proposed for code reuse attacks, too, which will be covered next.

Question?

Exercise

Find a syscall ROP gadget from binaries on your machine. Can you easily spot it?