Reverse engineering and binary exploitation applied to OPIChat

ACU 2022 Team

Copyright

Introduction

Fuzzing

- You saw basic concepts of fuzzing during your last project, OPIChat
- The goal is to search for security vulnerabilities
- But is a segfault a serious security issue?

- Understand inner working of compiled binaries
- Exploit design flaws in code to inject and run your own
- From segfault to arbitrary code execution
- Transform your OPIChat into a Minecraft server

Legal warning

- Exploiting someone else's information system is illegal
- Always have written approval before doing so
- Check a program license to see if you are authorized to reverse engineer it

- Understand how a system works with limited information about how it does so
- For example: understand a compiled binary without the source code
- Main use cases:
  - Vulnerability research
  - Malware analysis
  - Industrial espionage

Static vs dynamic analysis

- Static analysis: Understand a binary without running it
  - Disassembler (`objdump`, `radare2`, `IDA`, ...)
  - Decompiler (`IDA`, `binary ninja`, ...)
  - Static code analysis
- Dynamic analysis: Study a program during its execution
  - Debugger (`gdb`)
  - Tracer (`strace`, `ltrace`, ...)
  - Network sniffers (`tcpdump`, `wireshark`, ...)
  - VM

ELFs

Programs

- Programs are files
- Contain both data and code

ELF

- `Executable and Linkable Format`
- Format for `UNIX` executable files
- Contains sections to represent different types of data

Sections

- `.data`: global variables
- `.rodata`: read only data (e.g. string literals)
- `.got`: contains dynamic libraries functions pointers
- `.text`: contains executable code

Process: calling `execve(2)`

- A process is a running instance of a program
- `execve(2)` maps pages and loads the sections
- Start execution at the entrypoint

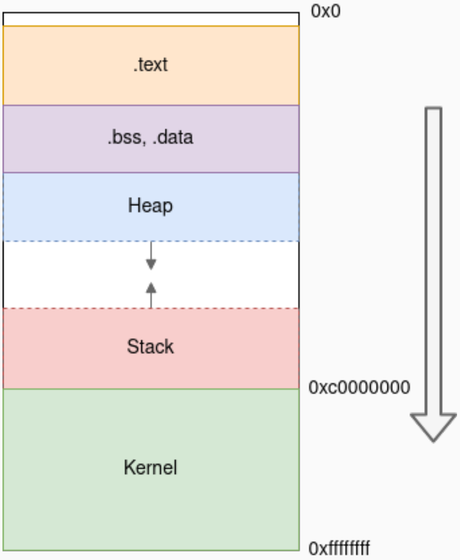


Figure 1: Process address space

Stack

- Contains:
  - Function stack frames
  - Local function variables
  - Environment variables

.text

- A special section containing the compiled, executable machine code
- Binary data corresponding to processor instructions
- Bijection with assembly language

x86 Assembly

Concepts

- Human readable transcription of machine code
- Ordered list of instructions which can take up to 3 operands
- Operands can be literal values, memory accesses or registers

Example

```c
int main(void)
{
    int a = 4;
    for (int i = 0; i < 38; ++i)
        ++a;
    return a;
}
```

```
main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $8, %rsp
    movl    $4, -8(%rbp)
    movl    $0, -4(%rbp)
    jmp    .L2
.L3:
    addl    $1, -8(%rbp)
    addl    $1, -4(%rbp)
.L2:
    cmpl    $37, -4(%rbp)
    jle     .L3
    movl    -8(%rbp), %eax
    popq    %rbp
    ret
```

Registers

- Registers are fast memory storages inside the processor
- General registers, used to perform most general computing instructions (additions, multiplications...):
  - `eax`
  - `ebx`
  - `ecx`
  - ...
- Special registers:
  - `esp`: Stack pointer
  - `ebp`: Frame pointer
  - `eip`: Instruction pointer

`esp`: Stack pointer

- Pointer to the current top of the stack
- Allocating local variables and calling functions decreases it
- Returning from functions increases it

- The `push` instruction decrements `esp` and stores its operand in it
- The `pop` instruction is the inverse: stores the data pointed by `esp` in its operand and increments it

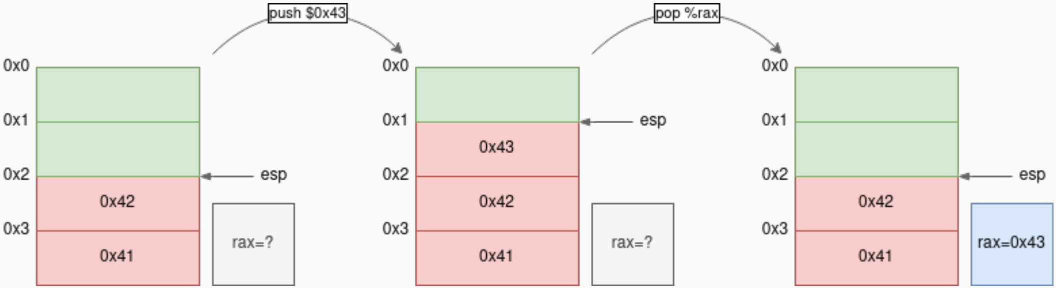


Figure 2: Push instruction

`eip`: Instruction pointer

- `eip` is a 32 bits register
- Indicates to the processor the address of the next instruction to be executed
- Is incremented automatically after executing an instruction
- Altering it also alters the execution flow of the process

Control flow

Many instructions can alter `eip`

- `jmp`: Unconditional jump
- `je`, `jne`, `jge`, `jl`, ...: Conditional jumps
- `call`: Function call
- `ret`: Function return

Functions in assembly

- `call` and `ret` are inverse operations
- `call addr` is actually the succession of:
  - `push %eip` to save the current instruction
  - `jmp addr` to execute the function
- `ret` is just a `pop %eip`, effectively restoring `eip` to its value before the call

Stack frames

- A stack frame is the context of a function (i.e. its local variables)
- The lifetime of a stack frame is the same as the function
- Calling a function pushes a stack frame
- Returning from a function pops the stack frame and restores the previous one
- Initializing and restoring stack frames is called the `prologue` and the `epilogue` of a function

Prologue and epilogue

- `ebp` is the frame pointer, it points to the start of the current stack frame
- The end of the stack frame is `esp`
- Everything in between are local variables of the function

Prologue

- To create a new stack frame, the prologue must:
  - `push %ebp` to save the start of the previous stack frame
  - Move the current value of `esp` in `ebp`
  - Decrease `esp` to allocate the stack frame

Epilogue

- To restore a stack frame, the epilogue must:
  - Move `ebp` in `esp` to deallocate the stack frame
  - `pop %ebp` to restore the start of the previous stack frame

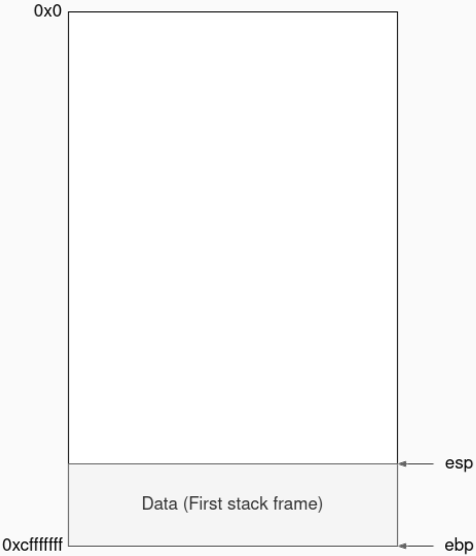
Recap

```
function:
    pushl   %ebp        # Save previous stack frame
    movl    %esp, %ebp  # Init new stack frame
    subl    $42, %esp   # Allocates 42 bytes in the stack frame for local variables

    ...                 # Function code

    movl    %ebp, %esp  # Deallocate stack frame
    popl    %ebp        # Restore previous stack frame
    ret

main:
    call function
```

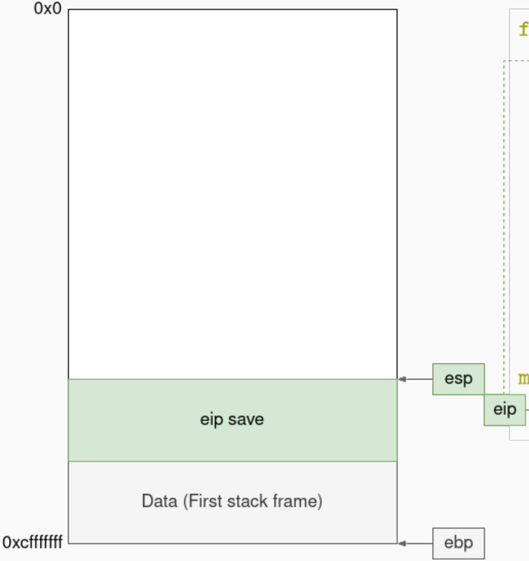



```
function:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $42, %esp

    ...

    movl    %ebp, %esp
    popl    %ebp
    ret

main:
    call function
```

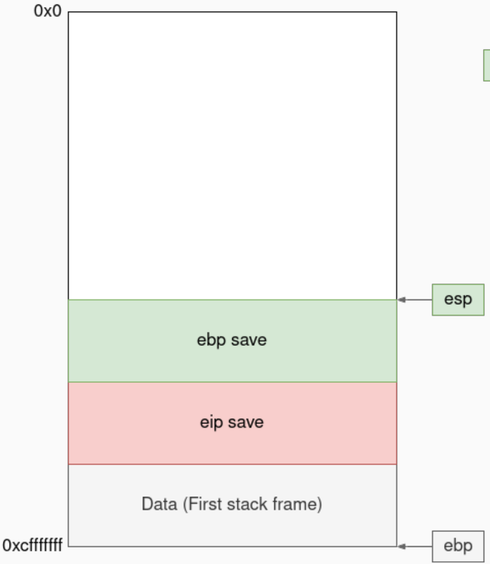



```
function:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $42, %esp

    ...

    movl    %ebp, %esp
    popl    %ebp
    ret

main:
    call function
```



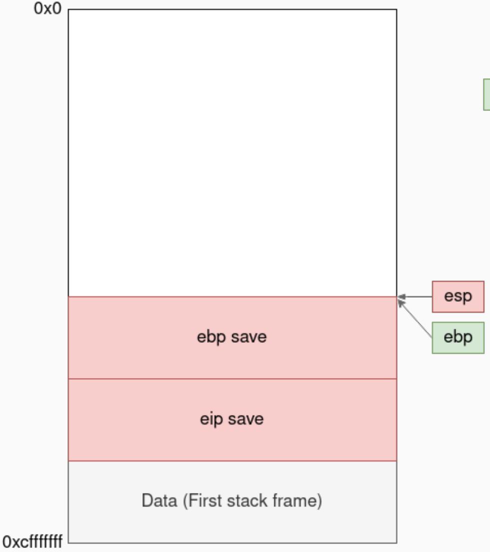
eip →

```
function:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $42, %esp

    ...

    movl    %ebp, %esp
    popl    %ebp
    ret

main:
    call function
```

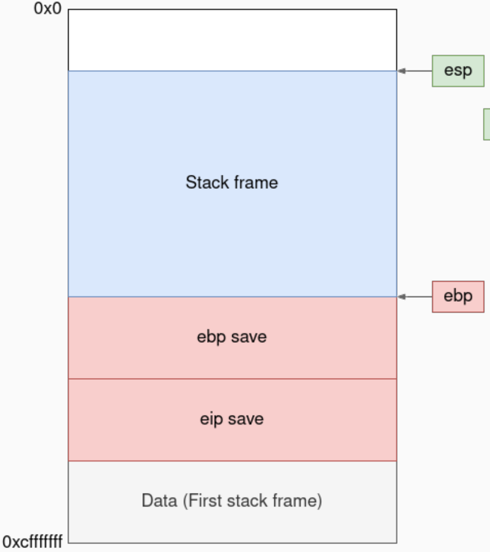



```
function:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $42, %esp

    ...

    movl    %ebp, %esp
    popl    %ebp
    ret

main:
    call function
```

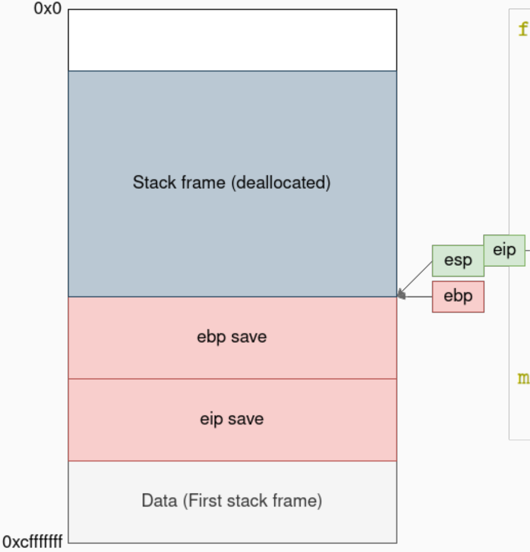



```
function:
    pushl   %ebp
    movl    %esp, %ebp
eip → subl  $42, %esp

    ...

    movl    %ebp, %esp
    popl    %ebp
    ret

main:
    call function
```

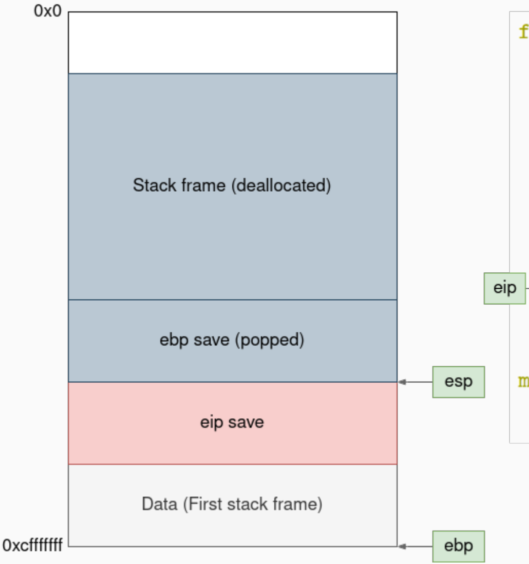



```
function:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $42, %esp

    ...

    movl    %ebp, %esp
    popl    %ebp
    ret

main:
    call function
```

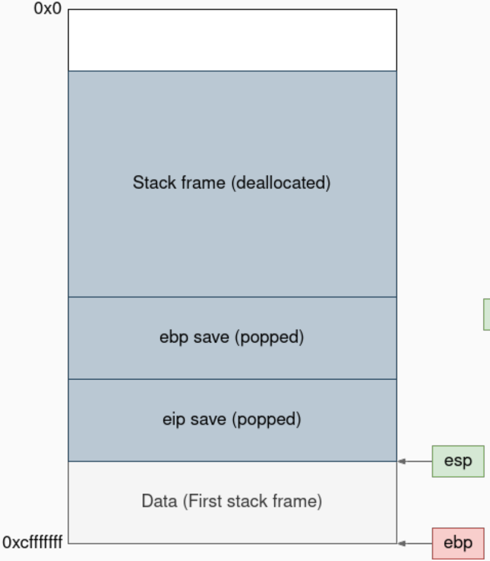



```
function:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $42, %esp

    ...

    movl    %ebp, %esp
    popl    %ebp
    ret

main:
    call function
```




```
function:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $42, %esp

    ...

    movl    %ebp, %esp
    popl    %ebp
    ret

main:
    call function
```

Execution flow hijacking

`eip` control vectors

- `eip` is a critical register
- Controlling it means controlling the whole execution flow of the process
- It is updated automatically by the processor
- Few instructions allow to modify `eip` with values hardcoded in `.text` and therefore not controllable by an attacker
- Except one

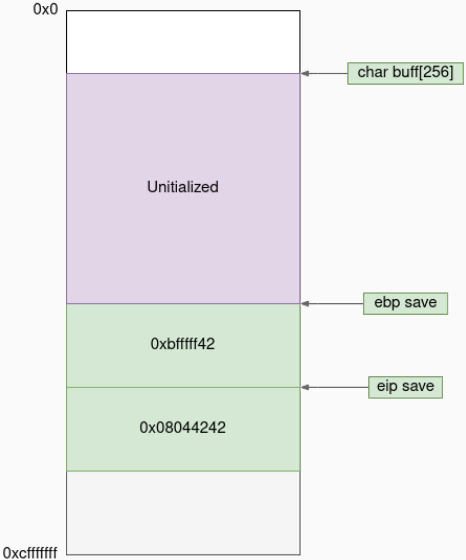
`ret`

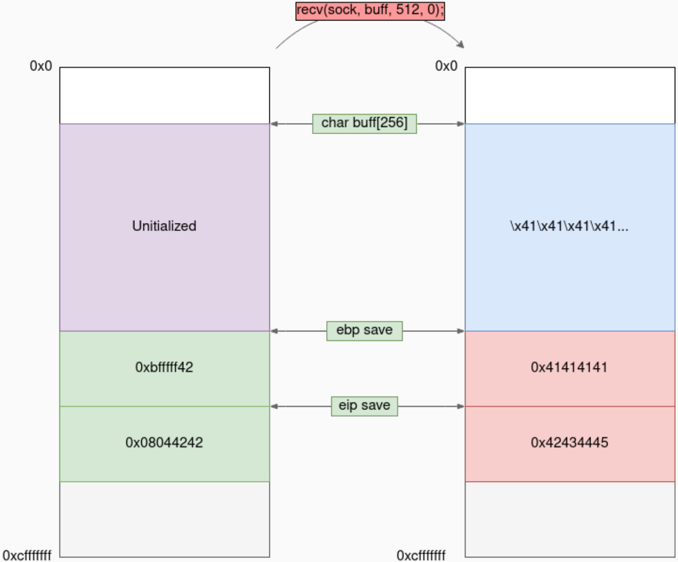
- `ret` recovers the return address from the stack (a writeable segment)
- If an attacker can control the `eip` save, they can redirect the execution flow of the process wherever he wants

Stack buffer overflow: my first vulnerability

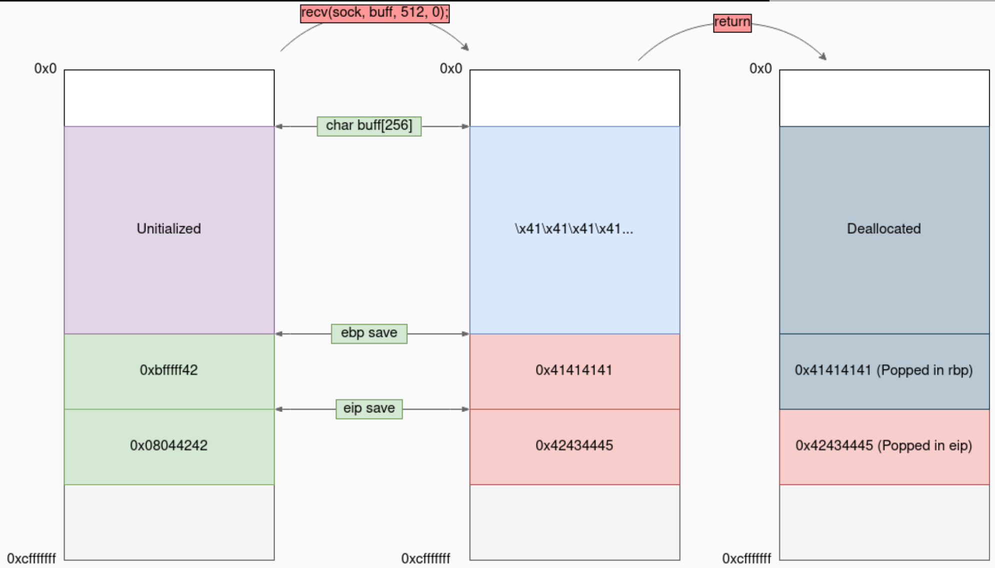
- One of the most simple and critical vulnerability
- Caused by writing data in a stack buffer smaller than the data

```c
void vuln(int sock)
{
    char buff[256];
    recv(sock, buff, 512, 0);
    return;
}
```


0x0
char buff[256]
Unitialized
ebp save
0xbfffff42
eip save
0x08044242
0xcfffffff



- Suppose an attacker sends 260 ‘A’ followed by ‘EDCB’



- `eip` now has value `0x42434445`, effectively redirecting execution to this instruction

Demo execution flow hijacking

Arbitrary code execution

Arbitrary code execution

- We can now redirect the execution flow wherever we want
- But where can we jump?
- Basically any executable segment
- Which might be our stack!

Stack code injection

- If there is valid x86 code in the stack, we can jump on it
- But wait, we control part of the stack since the request is received in the stack!
- So we can also inject code in our request

Shellcodes

- A shellcode is a small piece of machine code easily injectable whose aim is generally to execute a shell (thus the name)
- We must inject this shellcode in the memory of the process (in our case in the stack)
- Then redirect `eip` to the shellcode to start executing our code

Easily injectable

In our case, an injectable shellcode must be accepted by the request parser and must not contain `\0` bytes.

Demo shellcode injection

Buffer overflow protections

Protections

- Canaries
- ASLR (Address Space Layout Randomization)
- Non-executable stack

Canaries

- Compilers put a value at the beginning of each stack frame
- In the epilogue, check if the value was modified
- `SIGABRT` if so
- Effectively checks if a you tried to rewrite data out of your stack frame

Canaries bypass

- Information leak
- Overflow “over” the canary

ASLR

- Randomizes the exact address of the start of stack
- Practically impossible to hardcode a return address in the stack since the shellcode will be at a different address every time

ASLR bypass

- Information leak
- RET2REG
- pop ret
- ret to libc
- ROP

Non-executable stack

- Remove the executable permission of the stack
- trying to `ret` to the stack will `SIGSEGV`

Non-executable stack bypass

- ret to libc
- ROP

Demo: find the vulnerability

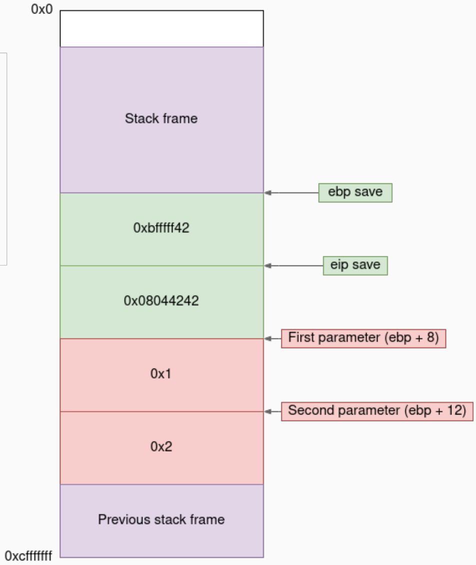
Calling conventions

- Function parameters are pushed on the stack in reverse order

```c
void func(int a, int b);

int main(void)
{
    func(1, 2);
}
```

```asm
pushl   $2
pushl   $1
call    func
```



Format string bug

- Occurs when an attacker controls a `printf` format string

```
printf(controlled_by_attacker);
```

- They can inject `printf` directives (e.g. `%x`, `%p`, ...)
- Every `printf` directive will try access a parameter that was not pushed
- Thus effectively reading data from the stack

%n: Transform printf into a Turing machine

- The `%n` directive takes the corresponding variadic argument as an `int *`
- Writes the number of bytes printed by printf until now into the address pointed by the variadic argument
- If the attacker also controls part of the stack, he can inject custom printf variadic arguments, thus giving arbitrary memory address to `%n`
- By carefully writing bytes (e.g. with the `%c` directive), it is possible to write arbitrary data anywhere in the memory

Demo format string bug

Format string protections

Protections

- Non-executable stack
- PIE
- RELRO

PIE

- Position Independant Executable
- Similar to ASLR but for `.text`
- Makes it practically impossible to jump in the existing code

RELRO

- RELocation Read-Only
- Dynamic library functions pointers are read-only and cannot be modified at runtime
- We cannot redirect `libc` calls anymore

Conclusion

Post exploitation

- Privilege escalation
- Lateralisation/Persistence
- Obfuscation

Tools used

- peda: a gdb plugin for reverse engineering and exploit development
- objdump: command line disassembler
- binary ninja/IDA: advanced disassembler/decompiler
- cutter/radare2/ghidra: Free alternatives to binary ninja