Introduction to software exploitation

ISSISP 2017

VM

https://drive.google.com/open?id=0B8bzF4YBus1kLTJSNlNWQjhsS1E (sha1sum: 36c32a596bbc908729ea9333f3da10918e24d767)

Login / pass: issisp / issisp

Who are we

- Josselin Feist, josselin@trailofbits.com
- Mark Mossberg, mark@trailofbits.com
- Trail of Bits: trailofbits.com
  - Help to build safer software
  - R&D focused: use of the latest program analysis techniques

Plan for Today

- Basic concepts of software exploitation
  - What is a buffer overflow
  - How to exploit it
- Two hands-on:
  - Simple buffer overflow to exploit, using debugger
  - More complex example, using symbolic execution

Program Vulnerabilities

- Programs contain tons of bugs
  - Some are benign
  - Some impact the security of the system: vulnerabilities
- How to find them:
  - Manual inspecting
  - Fuzzing
- Use of a vuln to corrupt the system = exploitation

Software Exploitation

- Why does it matter?
  - Attack: obvious reasons
  - Defense:
    - Knowing if a vulnerability is exploitable -> prioritization
    - Help to convince developers to fix the vulnerability
  - Other reasons: CTF, interesting low-level manipulation, ...

Recall X64

Source Code Versus Assembly Code

- Programs usually written in high-level languages
  - C/C++, java, python, ..
- Compilation: Source code → binary
  - High-level code → assembly code
  - Variables → memory locations

Source Code Versus Assembly Code

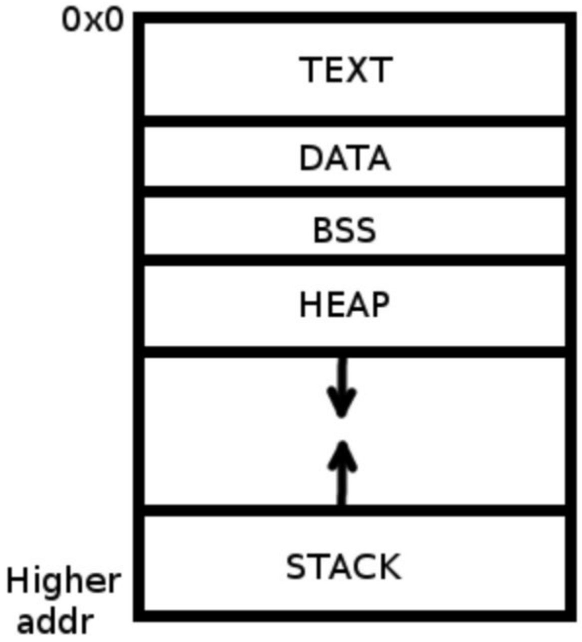
```c
#include <stdio.h>

int main(){
      printf("Hello world");
      return 0;
}
```

```
main:
00400526  push    rbp
00400527  mov     rbp, rsp
0040052a  mov     edi, 0x4005c4  {"Hello world"}
0040052f  mov     eax, 0x0
00400534  call    printf
00400539  mov     eax, 0x0
0040053e  pop     rbp
0040053f  retn
```

Program Variables

- Variables are split in sections:
  - **Local variable: stack**
  - Dynamic variable (malloc): heap
  - Others (constant, static,..) : data, rodata, ...



Program Variables

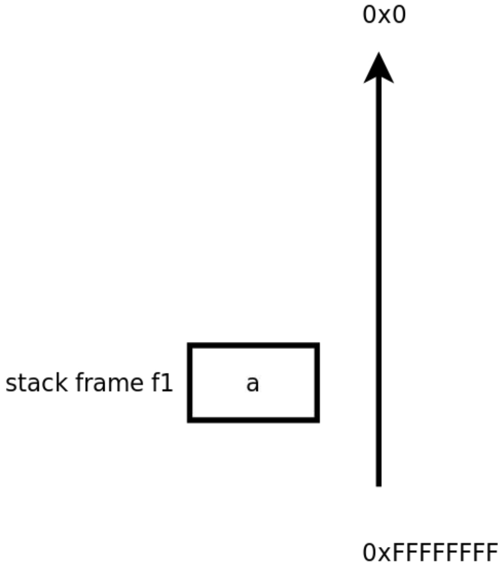
- Each function possesses its own ”**stack frame**”
- Stack is organized as LIFO
- It grows toward lower addresses (first element = highest address)

Stack Frame Example

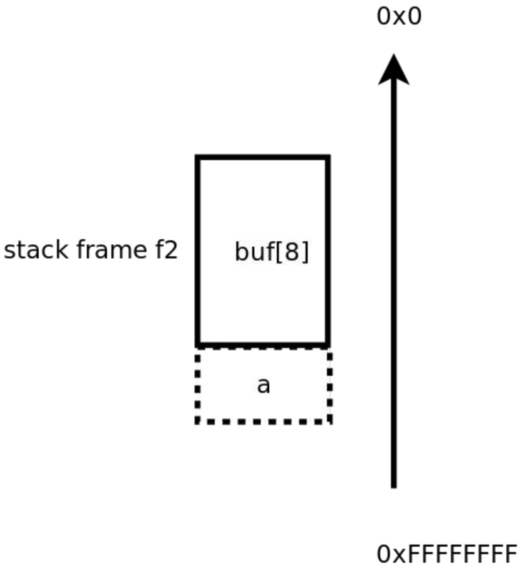
```c
void f2(){
 char buf[8];
}

void f1(){
 int a;
 f2();
}
```

Before f2() call



During f2()

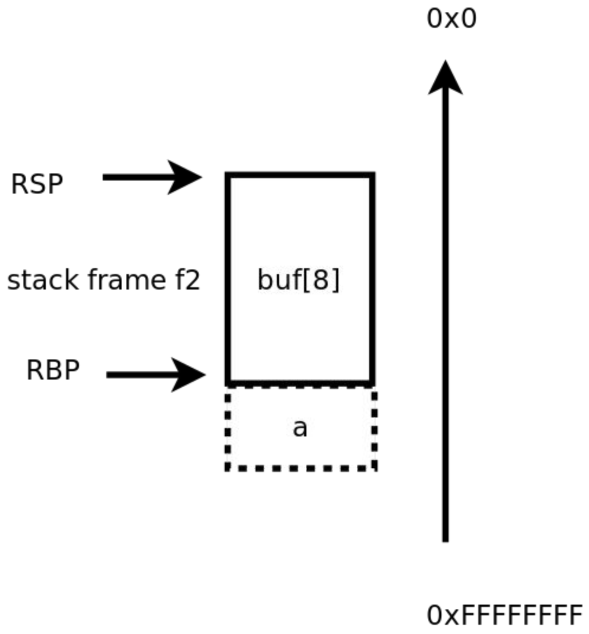


Stack Frame Example

Two specific registers: RSP and RBP

```c
void f2(){
  char buf[8];
}

void f1(){
  int a;
  f2();
}
```

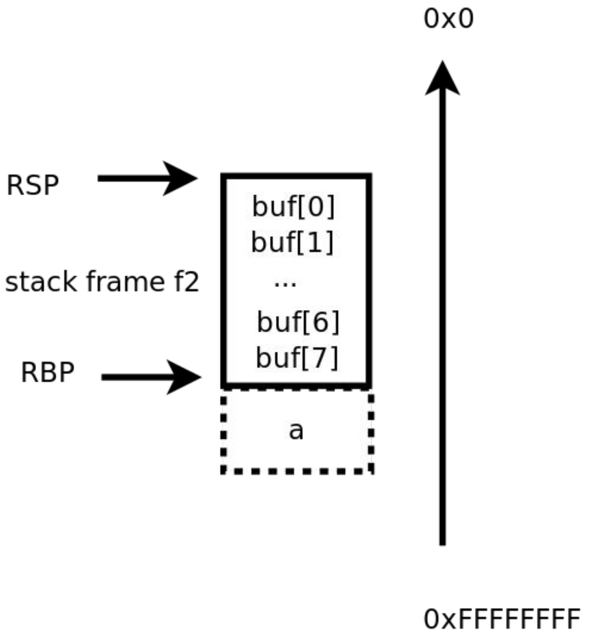


Stack Frame Example

Array elements grow toward higher values (@buf[0] < @buf[1])

```c
void f2(){
 char buf[8];
}

void f1(){
 int a;
 f2();
}
```



Stack Frame: Other usages

- The stack is used to store other elements
  - Function parameters
  - Saving registers during call: RBP and RIP
- Special register: RIP
- RIP points to the code that will be executed
- When a function returns, RIP needs to know where to return

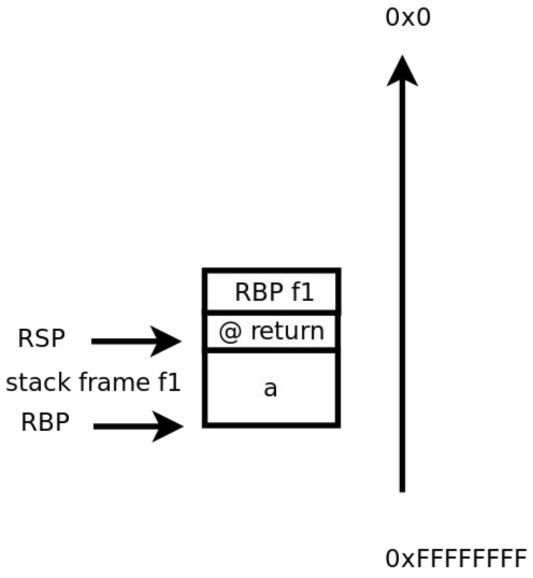
-> The stack stores data used for the control flow execution

Stack Frame Example

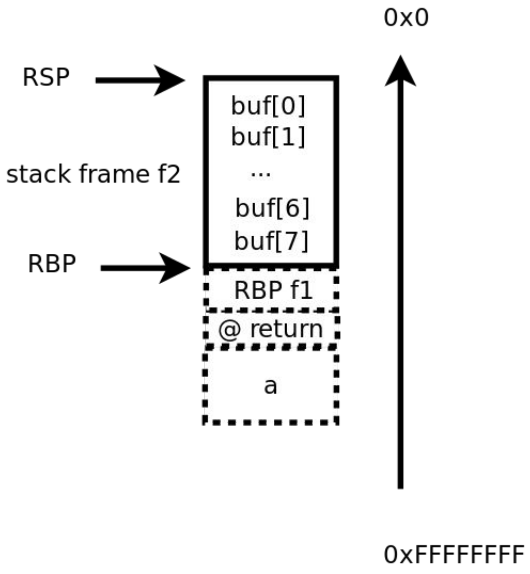
```c
void f2(){
  char buf[8];
}

void f1(){
  int a;
  f2();
}
```

When f2() is called



During f2()



Buffer Overflow

Stack-Based Buffer overflow

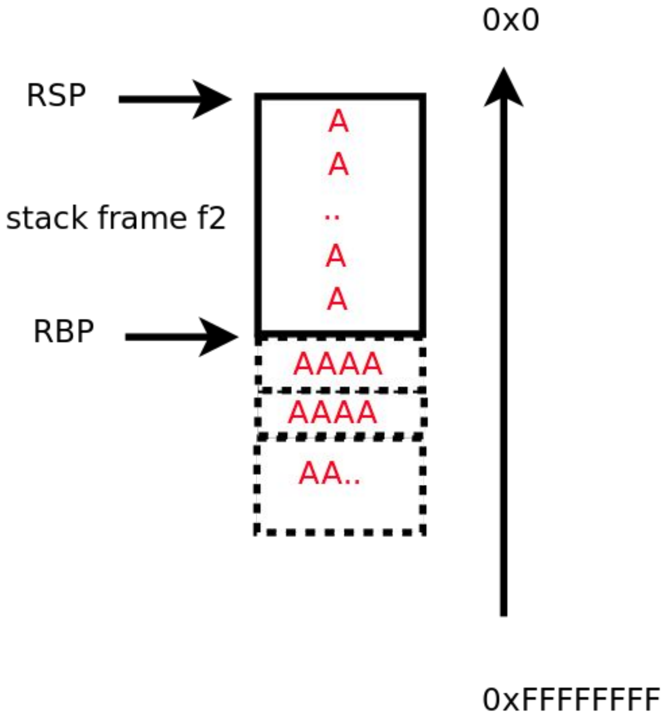
- If we write more than 8 elements in buf, we overwrite the stack, and thus the stored values
- When it happens:
  - Call to unsafe functions: strcpy, …
  - Call safer functions with a wrong size
  - Wrong number of loop iterations
  - ...

Buffer Overflow Example

Input = ‘AAAAAAAAAAA...AAA\o’

```
void f2(){
  char buf[8];
  strcpy(buf,input);
}

void f1(){
  int a;
  f2();
}
```



Control-Flow Hijacking

- The overflow rewrites the stored value of RIP
- You control RIP when the program returns
- Redirect the program execution flow wherever you want:
  - Usually, use of shellcode = small assembly code executing specific action (reading/writing file, …)
  - Goal for today: execute a specific function

Your goal

- Exploit the binary: /home/issisp/desktop/exo1/bof
- The subject: /home/issisp/desktop/exo1/subject.pdf

Goal: execute the function ‘print_secret’

Modern Exploitation

- Lots of protections against vulnerabilities:
  - Canary: a random value is put between stack frames, check if it is changed during execution
  - DEP: the stack is no longer executable (harder to use shellcode)
  - ASLR: sections are randomized
- In modern OS, you find even more complicated protections (EMET,...)

Second binary

Second binary

- $ cat crash.txt
  1AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
- $ ./vuln input.txt
  Segmentation fault (core dumped)
- $ gdb ./vuln
  run input.txt

=> 0x400bfe: call rdx

RDX: 0x4582c3004582c300

Crash Analysis

- Not a crash on the return of a function
- call rdx, with rdx = strange value
  - Direct link between the value of rdx and the input not explicit
- Not trivial to know the root cause
  - Call to a direct user-controlled value?
  - Buffer overflow leading to rewriting function pointer?
  - Other vulnerability? (Use-after-free, ..)

Crash Analysis

- One common solution: reverse-engineer the binary to understand the relation between the input and rdx

- The solution presented here: using dynamic symbolic execution to build the exploit

Dynamic Symbolic Execution

Dynamic Symbolic Execution (DSE)

- DSE: an automated input generation technique.

- Key idea: execute the program, but consider some variables as symbolic

DSE Example

```c
void f(int a){
 a = a+1;
 if(a == 0x42){
  printf("Win!\n");
 }
}
```

DSE Example

```
void f(int a){
 a = a+1;
 if(a == 0x42){
  printf("Win!\n");
 }
}
```

a is symbolic, called a0

DSE Example

```c
void f(int a){
 a = a+1;
 if(a == 0x42){
  printf("Win!\n");
 }
}
```

a is symbolic, called a0

a1 := a0 + 1

DSE Example

```c
void f(int a){
 a = a+1;
 if(a == 0x42){
  printf("Win!\n");
 }
}
```

a is symbolic, called a0

a1 := a0 + 1

Two possibilities:

- a1 == 0x42
- a1 != 0x42

DSE Example

```c
void f(int a){
  a = a+1;
  if(a == 0x42){
    printf("Win!\n");
  }
}
```

a is symbolic, called a0

a1 := a0 + 1

Two possibilities:

- a1 == 0x42
- a1 != 0x42

Two paths, represented as so-called path predicates:

- a1 := a0 +1 ^ a1 == 0x42
- a1 := a0 +1 ^ a1 != 0x42

Path Predicate

- Once you represent a path as a path predicate:
  - Ask a solver to give a valuation of symbolic inputs
    - Generating the inputs of the path
    - Proof that the path is not feasible
  - Add new constraints on the path predicate
    - Invert a condition
    - Force specific value (e.g. buf[i], i can be > size of buf[]?)

DSE

- Large recent interest in security
- Academic & industrial interest
  - Angr, Binsec, KLEE, Mayhem, SAGE, Triton, etc.
  - Today: Manticore
- Young topic, still a lot of limitations
- Different use:
  - Path exploration
  - Crash analysis
  - Deobfuscation
  - ...

Manticore

- Dynamic Binary Analysis Tool
  - Symbolic Execution
  - Taint Analysis
  - Program Instrumentation
- CLI Tool/Python API
  - Generate inputs
  - Query satisfiability
  - Script custom analyses
- x86/64, ARMv7

github.com/trailofbits/manticore

$ pip install manticore

Second binary (cont.)

Your goal

- Use Manticore to know if you can exploit the crash to call the function ‘print_secret’
  - You need an input leading to “rdx == @print_secret”