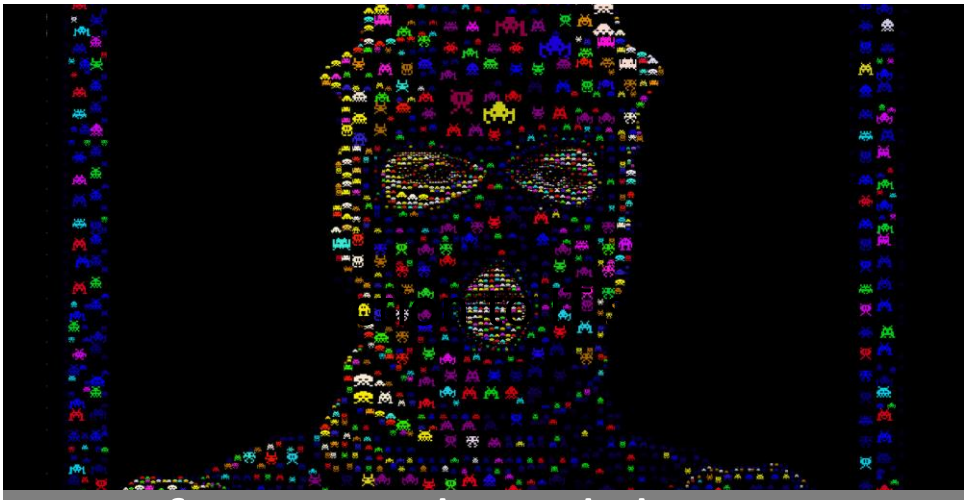


Software vulnerabilities p.1

Moshe Kravchik
Credits: Mark Stamp, Tzachy Reinman, Dan Boneh

https://flic.kr/p/kXT13r

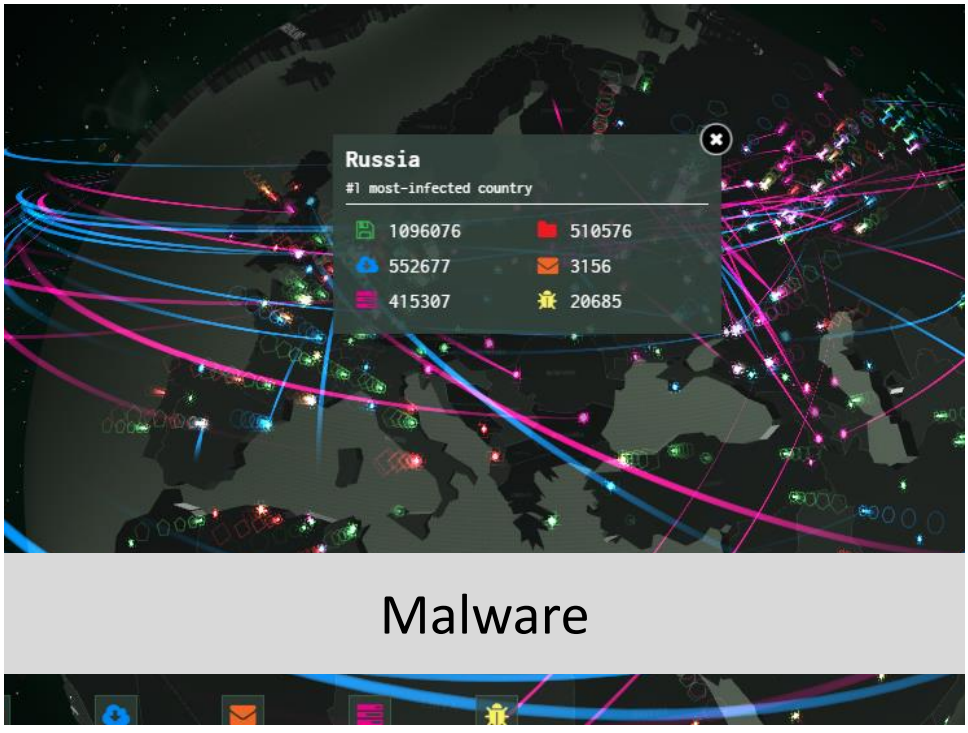
1

Agenda

- Malware and its kinds
- Buffer overflow
  - How it works
  - What can you do to prevent it

https://flic.kr/p/6keSjR

2



Malware

3

Why malware?

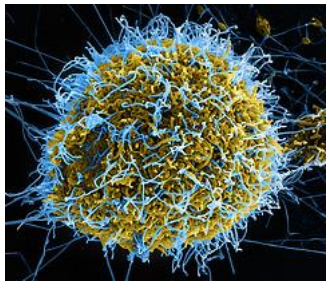
- Software is the most dynamic and powerful part of computer systems
- Taking over it means controlling the system
- Software has bugs and flaws…
- Malware exploits them
- Threatens PC, mobiles, networks, SCADA, …
- And costs fantastic amounts of money to customers

https://flic.kr/p/dMKYeW

4

Software vulnerabilities

- Program flaws (unintentional)
  - Memory management bugs
  - Logical errors
  - Race conditions
- Malicious software (intentional)
  - Viruses
  - Worms
  - Other breeds of malware

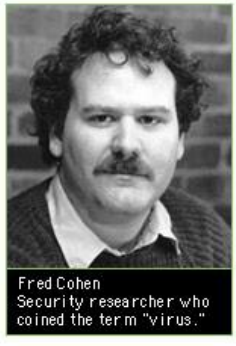


https://flic.kr/p/o15Y5n

5

Malware types

- Malware is not new…
  - Fred Cohen's initial virus work in 1980's, used viruses to break MLS systems
- Types of malware
  - **Virus** — passive propagation (requires user help)
  - **Worm** — active propagation
  - Trojan horse — unexpected functionality
  - Trapdoor/backdoor — unauthorized access



6

Software flaws

- Every software has some bugs…
- They are unintentional …
- But dangerous
  - Toyota unintended acceleration kills 89
  - Therac-25 radiation therapy machine overdoses 6 people
  - 1 billion $ European Space Agency's Ariane 5 Flight 501 self-destroys due to a software bug
- And can be exploited!

7

Most common vulnerabilities

- Memory exploitations
  - Buffer overflow
- Incomplete input validation
  - SQL injection
- Exploiting trust
  - XSS
- Logic errors
  - Integer overflows
- Race conditions (TOCTOU)

https://flic.kr/p/d5EFmq

8

Buffer overflow

- Was known in theory since 70s
- Used by the Morris worm in 1988
- Became widely popular after a legendary "Smashing The Stack For Fun And Profit" tutorial
- Used in many exploits
  - Code Red and SQL Slammer worm
- Stack overflow is the most famous form of it

https://flic.kr/p/fF7yF

9

Buffer overflow

```
int main(){
    int buffer[10];
    buffer[20] = 37;}
```

Q: What happens when code is executed?

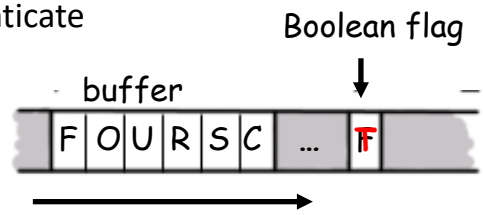
A: Depending on what resides in memory at location "buffer[20]"

- Might overwrite user data or code
- Might overwrite system data or code
- Or program could work just fine

10

Buffer overflow exploitation

- Consider boolean flag for authentication
- Buffer overflow could overwrite flag allowing anyone to authenticate

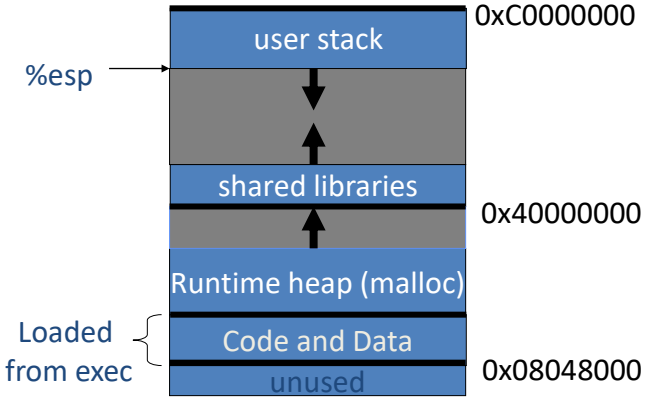


- But what attackers are looking for is code execution!
- Running arbitrary code with hacked program credentials

11

Process memory organization

Linux process virtual memory



12

Process stack

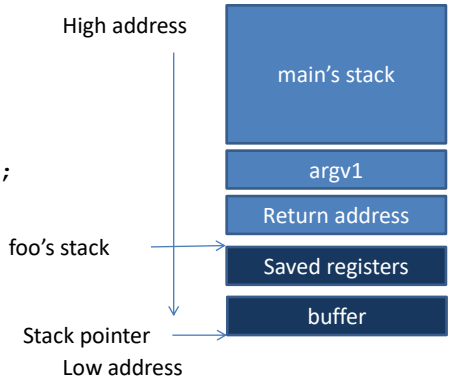
- A memory area used for local variables
- Used for passing the arguments between functions
  - Per thread
- Holds the return address
- Return address controls where the code execution will continue

https://flic.kr/p/9aKA3L

13

Function stack layout

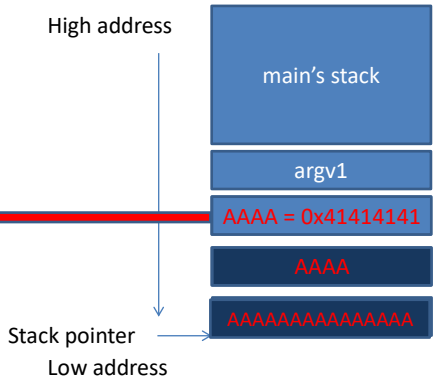
```
void foo(char *str)
{
      char buffer[16];
      strcpy(buffer,str);
}

void main(int argc,
char **argv)
{
      foo(argv[1]);
}
```



14

Stack smashing

bfbd2000-bfbf3000	stack
b7552000-b7768000	Shared libs
09d8a000-09dab000	heap
08048000-0804b000	Program




```
void foo(char *str)
{
        char buffer[16];
        strcpy(buffer,str);
}
```

- `stack_smashing("AAAAAAAAAAAAAAAAAAAAAA")`
- Segmentation fault (core dumped)

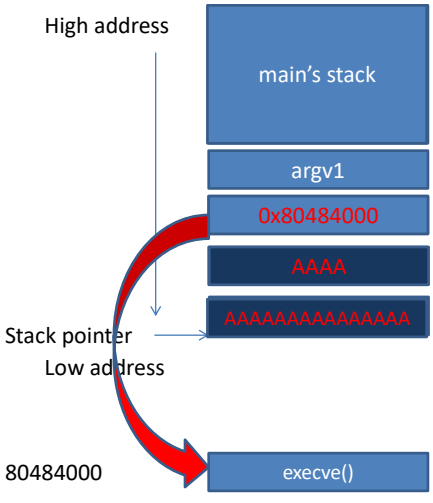
15

Stack smashing

- You can redirect code execution!

```
void main(int argc,
char **argv)
{
if(…)
        execve("/bin/sh")
}
```

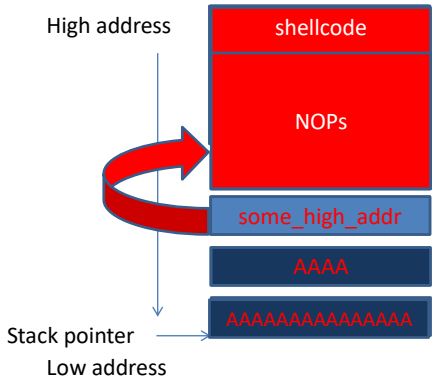
- Return-to-libc!



16

Stack smashing

- You can inject our own code!
- Writing shellcode is for another course!
- How do you guess the exact address?
- Use NOP-slide!



17

Buffer overflow dangers

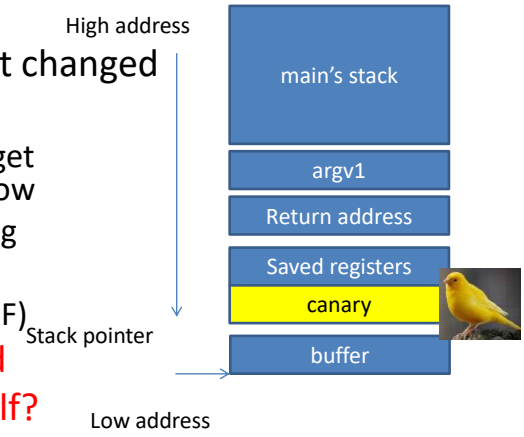
- The attacker can inject own code and run on behalf of the program
- Can be done remotely!
- The vulnerability can be found either by source code analysis (think open source)
- Or by binary reversing
- … or by fuzzing (try all possible inputs)

https://flic.kr/p/ebzM1d

18

Buffer overflow compiler protection

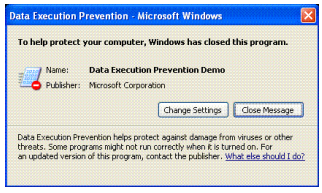
- Verify the stack hasn't changed
- Canary (cookie)
  - Add a value that will get overwritten by overflow
  - Verify before returning
  - Random
  - Use terminator (0, EOF)
- Requires code rebuild
- Overwritten with itself?
- Heap overrun?



19

Buffer overflow OS protections

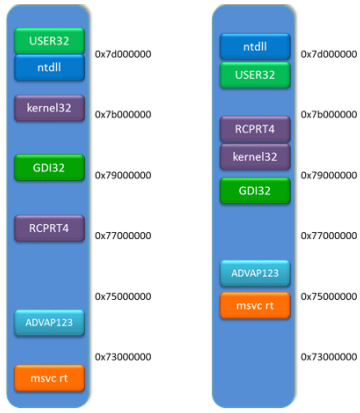
- Non-executable stack (and heap)
  - NX bit processor feature
  - All major OSes support it
- DEP (Data Execution Prevention)
  - Any data can't become executable
- Code Signing on iOS
- Some execution environments need to generate code in runtime (Javascript)
- Does not protect from return-to-libc



20

Buffer overflow OS protections

- Make it hard to build reliable exploits
- ASLR – Address Space Layout Randomization
- Load code, heap, stack and standard libraries at random addresses
- Return-to-libc becomes a problem
- In practice, there are some “not random” parts
- The amount of randomness is small (256 trials on 32 bit)
- Requires **code recompilation** to be position independent



21

Buffer overflow prevention

- Memory safe languages
- C#, Java, Python - all check for boundaries before accessing the memory
- Due to performance reasons C/C++ might be the only option
- There’s still lots of useful code running in native (C/C++)
- On many embedded devices you can only run native code

Microsoft® .NET

Java

python

22

Buffer overflow prevention

- What you can do as a **developer**?
  - OpenBSD example
  - Graceful failure
- Check for enough space
- Use safe C functions
  - str**n**cpy instead of strcpy
  - strcat_s and strcpy_s – safe, ~~but MS only~~ NOW PART OF C11 standard!!

23

Preventing Buffer Overflow (1)

With `strcpy` – buffer overflow is possible

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
  char buffer[10];
  if(argc < 2) {
      fprintf ( stderr, "USAGE: %s string\n",
                argv[0]);
      return 1;
  }
  strcpy(buffer, argv[1]);
  return 0;
}
```

Input validation missing!

24

Preventing Buffer Overflow (2)

Use `strncpy` – buffer overflow is prevented

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
  char buffer[10];
  if(argc < 2) {
     fprintf ( stderr, "USAGE: %s string\n",
               argv[0]);
     return 1;
  }
  strncpy(buffer, argv[1], sizeof(buffer));
  buffer[sizeof(buffer) - 1] = '\0';
  return 0;
}
```

- But why this zero at the end?

25

Preventing Buffer Overflow (3)

A closer look at `strncpy` – where is the second string written?

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
  char buffer[101];
  strncpy(buffer, argv[1], 10);
  strncat(buffer, argv[2], 90);
  return 0;
}
```

strncpy doesn't automatically null-terminate the string being copied into.
In the subsequent strncat, data is copied not to buffer[10] as the code suggests, **but to the first location to the left of buffer[0] that happens to contain a zero byte.**

26

Buffer overflow prevention

- Make sure to compile with stack protection (StackGuard, ProPolice, /GS)
- Compile your code as position independent so ASLR will be used
- Input filtering and sanitization
  - Exploits use multiple NOPs
  - Application-level firewall
  - We'll talk more about input validation…

https://flic.kr/p/K3vrY

27

29

Terms learnt

- Malware
- Virus, worm, trojan
- Overflow
- Stack smashing
- Return-to-libs
- Canary
- Stack protection
- DEP, ASLR

30

30

Summary

- Malware is the biggest security threat
- It exploits bugs
- Stack overflow allows remote code execution
- Use OS and compiler protections
- Choose safe languages and libraries
- What can go wrong?

31