

Lecture #6

Ex 6 is on moodle

- Hard and relates to today's subject
- Let him know

Project submission box

- Upload the powerpoint

Homework is reading and doing the whole thing

ACL and C list

Take screenshots of the stuff with john the reaper

Buffer overflow

Malware

- Malicious software

Russia virus

Kaspersky anti virus company maps a lot of attacks

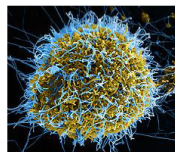


Malware

- Important part
 - o Software = computer brain
 - o Software has bugs – human factor
 - o Mistakes
 - Lazy or tired

Exploits on PC, mobile, networks, scada – controllers

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



<https://flic.kr/p/o1SY5n>

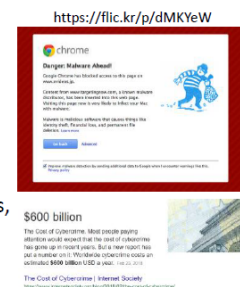
Worms spread to other computers

Worms = malicious software

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>



Lecture #6

Download worm to computer spreads by itself, knows vulnerabilities, exploits all possible ways to spread and infect other computers

- Active and reproduces

Viruses

- Passive attack
- Unless click file nothing will happen
- Doesn't spread by itself

Race conditions

Exploits management bugs

Bot – install via virus or worm

Zombie pc – infected pc – joins zombie network

Ademan part of RSA group – fred's teacher

Naïve programming – didn't think about viruses

Trojans – unexpected

Bugs = bad

Medical devices need software

Different vulns

- Buffer overflow relates to memory
- SQL injection

Time of check, time of use – race conditions

Buffer overflow

- Was known in theory since 70s
- Used by the [Morris worm](#) in 1988
- Became widely popular after the 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



Morris worm shut down internet

Levi – smashing stack

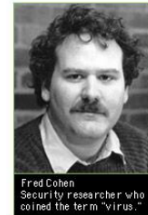
Malware types

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



Fred Cohen
Security researcher who
coined the term "virus."

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-destructs due to a software bug
- And can be exploited!



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>

Lecture #6

Famous attack is stack overflow – related to buffer overflow

Also have heap overflow

Writing to memory not allocated – overwrite something... user data, code etc.

Buffer contains password and flag to tell us if password is correct or not

User can go to the flag and change it to be true

Buffer holds 0 for false, anything else = true

How do you find spot? – don't know but can check via user system input till a result shows up

Looking for code execution, not just flipping a flag

Virtual memory

- 32 bit machine with 4gb
- OS keeps part of memory for itself
- Should be 8 times F

Shared libs = system libraries

Stack can grow down and shared libs are constant but still can grow up

Heap = dynamic memory allocation

- Use new keyword

Code and data at bottom

%esp points to end of stack

- Thread pointer
 - o Holds pointer of all threads run on system

Process stack

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



Local vars, pass variables and args between fns use stack

Buffer overflow

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

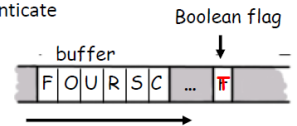
Q: What happens when code is executed?

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

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

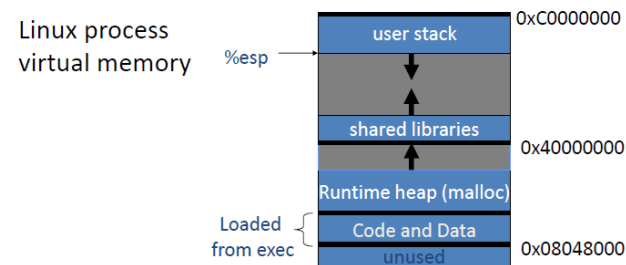
Buffer overflow exploitation

- 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

Process memory organization



Lecture #6

Per thread and holds \$ra (return address)

Computers have memory and register

Registers are much faster than memory itself

- We want to run our own programs
 - o OS prefers to give us dedicated program memory – stack not registers

FNs pass vars and arguments – kept in stack

After program is done running, OS will erase everything and start again

Also holds \$ra

Local var called buffer

Arg = # args

And pointer to arguments we get from user (size unknown)

- Changes each time

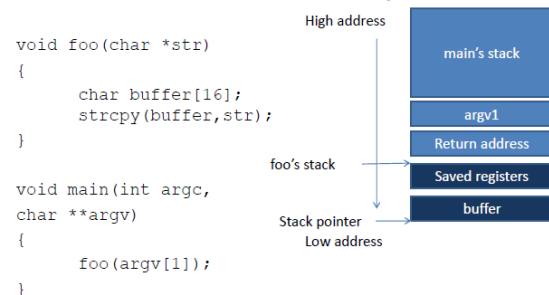
RA – line appears under foo

Where system continues from

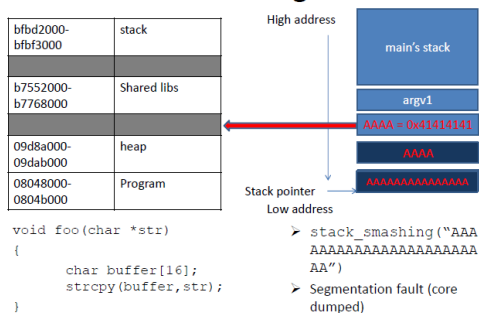
Save register – used by compiler

- Our fn and software wont use it

Function stack layout



Stack smashing - basic



15

How to exploit, enter more than 16 chars

- Fill buffer first
- Then fill save registers
- Last thing is return address which gets filled

Now...

Lecture #6

Computer will try to go to RA which now was overwritten, system most likely crashes

No meaning to location in \$ra

Compiler converts AAAA to hex because he expects to get an address there

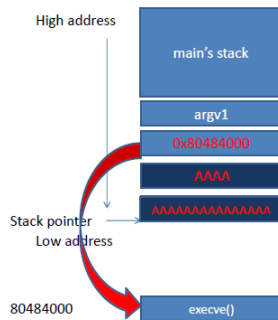
Another thing we can do:

Stack smashing – return to libc

- You can redirect code execution!

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

- Return-to-libc!



If he knows the location in memory of a specific function, he could go straight to the fn because he gives the ra to compiler to run

In this case its shared-code

Runtime c libraries

Gives lots of privileges to the user with /bin/sh

Return to libc attack – runtime c libraries

now buffer overflow but get code straight into the stack

Fill buffer, registers, and \$ra will be determined – not part of known function

To replace it with the main fn

Run own shell-code

Not so easy

They know stack boundaries but not exact location of RA

RA is in the stack

NOP – no operations

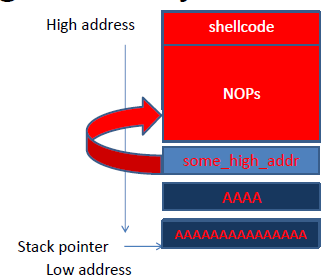
- Before put code, shell code (in assembly)

Put NOPs before shell code – when we get to RA, we will slide over NOPs

Fill memory with NOPS but now we get RA

Stack smashing – code injection

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



Nops continue but nothing happens till get to shell code which now runs what is inside to get privileges to hack the system

Adjust code per machine as assembly is different per architecture

OSS – easy to find these bugs

- Code is available

Windows find bugs?

- Microsoft devs stole code
- Cannot obscure code

Reverse engineering – binary reversing

Fuzzing – feed system with everything we get till we find a satisfactory result

Get paid by finding bugs

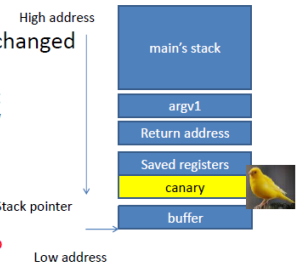
- The attacker can inject own code and run on behalf of the program
- Can be done remotely!
- The vulnerability can be found either by source code analysis (think open source)
- Or by binary reversing
- ... or by [fuzzing](#) (try all possible inputs)



<https://flic.kr>

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?



How to protect against buffer overflows

- Now compilers use canaries
- With miners – gas leak

Canary's alert if we have buffer overflow

- Random sequence generated by compiler
- Tell compiler to use canaries
 - **EX 6 – understand it better**

End of File – inside canary sequence

- Now even if attacker copies the sequence to continue

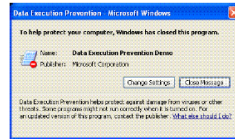
0 or EOF program will stop and not continue

Note: old code

- Hard to convert it or reprogram it – canary not option
- Not protect from heap overrun
 - Another unrelated issue related to memory

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
- No need to rebuild the code
- Code Signing on iOS
- Some execution environments need to generate code in runtime (Javascript)
- Does not protect from return-to-libc



Non executable stack

- We insert into stack strings of data

Strings tell compiler to do something – sequence of A's translate to address

Cannot execute comments on stack – but there is a catch

Can tell compiler if it is data, keep it as data DO NOT Interpret it nor execute it

Don't rebuild code, just use new memory designs

iOS code signing

- Get executed only if signed by a certified programmer

If there is a back door , you will did it

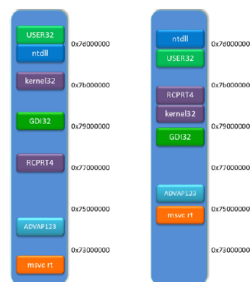
JS – dynamic environment

Code generated in run-time -> hard to sign the code

Runtime library code – libc use built in functions

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



Protect against lib-c: shuffle the libraries so attack wont know which fn relates to a location specifically and wont execute that fn because wont know location ASLR

Lecture #6

But still use brute force to find it

32 bit system need 256 tries to break it

Another way to protect stack

Using c and c++ cause these issues – language allows us to exploit it

Use a safer language

Protects us from memory bugs

No free lunch

C++ optimized

Python built on C – math libraries all in C

Use high level languages

OpenBSD – similar to unix

But devs went through system and debugged it system

Use safe C functions

Not all safe fns are supported by major OSes

another buffer overflow

- Copy string to the buffer with
- Missing input validation
- Strcpy copies each character doesn't care about arg length
- Easily get into buffer issue

Preventing Buffer Overflow

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 str:
                argv[0]);
        return 1;
    }
    strncpy(buffer, argv[1], sizeof(buffer));
    buffer[sizeof(buffer) - 1] = '\0';
    return 0;
}
```

- But why this zero at the end?

Buffer overflow prevention

Memory safe languages

C#, Java, Python, Rust - all check for boundaries before accessing the memory

Due to performance reasons C/C++ might be the only option (except for Rust)

There's still lots of useful code running in native (C/C++)

On many embedded devices you can only run native code



Buffer overflow prevention

- What you can do as a **developer**?

- OpenBSD example

- Graceful failure

- Check for enough space

- Use safe C functions

- `strncpy` instead of `strcpy`

- `strcat_s` and `strcpy_s` – safe, but MS only. NOW PART OF C11 standard!! ... but are not supported on Linux



so secure

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!

strcpy

and

\0 null terminator

Protect memory from attack

Send to argv a string longer than 10, it copies first 10 chars

Then strncat will try to attach last 90 chars to buffer – can give much longer string of course

But puts at end of buffer – unknown

Search for null or 0 and it could be in the memory

Not necessarily be at the end of the buffer

Probably garbage there

If we added null at end of buffer it would have stopped it

Long string will overwrite buffer and everything

Important to put null at end of buffer

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.

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?

Use smart compilers to avoid attack

