Information Security

System Security 1 - Memory Safety

November 11, 2022

Memory Safety

Memory safety - Wikipedia

Memory safety is a concern in software development that aims to *avoid software bugs* that cause security *vulnerabilities* dealing with random-access memory *(RAM) access*, such as buffer overflows and dangling pointers.

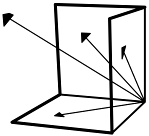
Memory Safety - More Details

A program execution is memory safe if the following things do not occur:

- Access errors
  - Buffer overflow/over-read
  - Invalid pointer
  - Race condition
  - Use after free
- Uninitialized variables
  - Null pointer access
  - Uninitialized pointer access
- Memory leaks
  - Stack/heap overflow
  - Invalid free
  - Unwanted aliasing

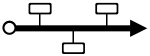
Memory Safety Violation

Two types of memory safety violation



Spatial violation: memory access is out of object's bounds

- buffer overflow
- out-of-bounds reads
- null pointer dereference

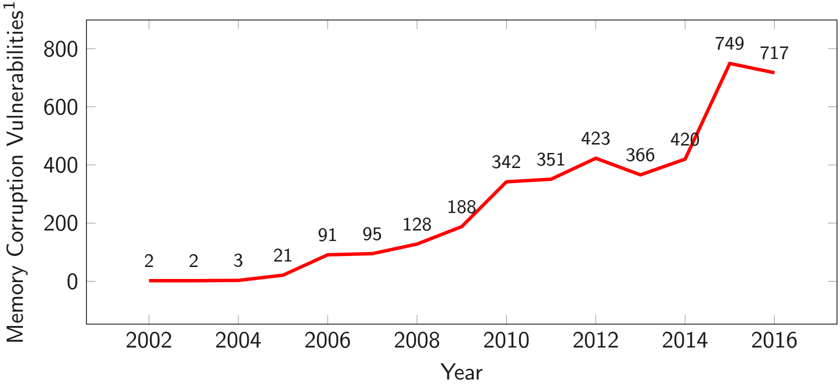


Temporal violation: memory access refers to an invalid object

- use after free
- double free
- use of uninitialized memory

Motivation

The complexer the programs, the more bugs



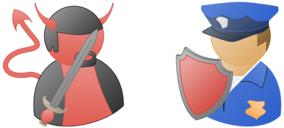
[1]Source: `http://www.cvedetails.com/vulnerabilities-by-types.php`

Red Team vs Blue Team

- There are two views on memory safety:
  - Attackers try to violate memory safety
  - Defenders try to ensure memory safety
- Attackers and defenders are often seen as teams in a “security war game”
- The Red Team tries to find security problems and mount attacks

- The Blue Team tries to protect software and defend against attacks

Red Team vs Blue Team



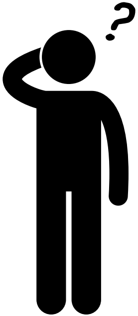
- The Red Team are not (only) criminals, their work is essential for the Blue Team
- Blue Team develops defenses based on Red Team attacks
- Red Team breaks them again

→ More secure software and better defenses

- Ultimate goal: memory safe programs

Red Team aka Attacks

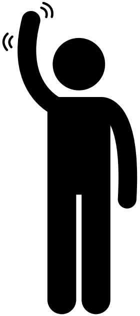
What is an Exploit?



- What is an exploit?
- “a software tool designed to take advantage of a flaw in a computer system” (Oxford)
- “[...] cause unintended or unanticipated behavior to occur on computer software” (Wikipedia)
- “If Achilles’s heel was his vulnerability in the Iliad, then Paris’s poison tipped arrow was the exploit. ” (Kaspersky)

→ Quite fuzzy

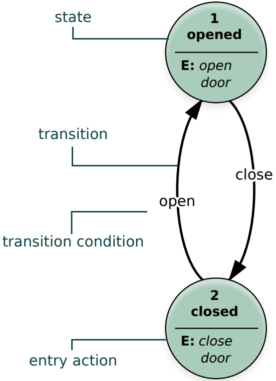
What is a “normal” program?[2]



- Programs: machines solving a certain problem(?)
- Ideally, finite-state machines
- We don’t build such machines → general-purpose hardware emulating them
- Programs: emulators for finite-state machines

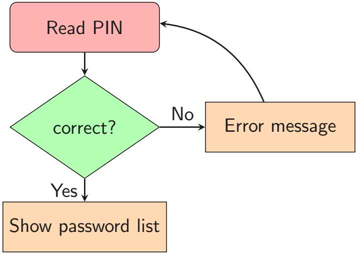
[2]Most of the following ideas are from Halvar Flake / Thomas Dullien

What is a “normal” program?



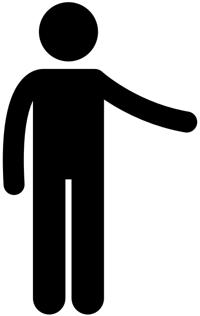
- Finite-state machines: states and transitions
- Input: changes state to different state
- Finite-state machine (FSM) solves your problem
- Many different ways to implement FSM

An Example: Simple Password Manager



- Security properties for your FSM
- Security properties based on inputs and outputs
- e.g., *It should be practically infeasible for an attacker to get the password list (output) if he does not know the PIN (input)*

Finite-state machine states vs CPU states

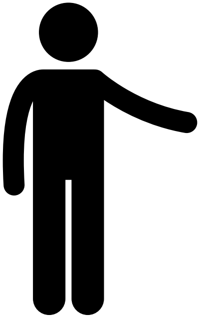


- We have to write an emulator for our FSM
- CPU has a lot more states than our FSM
- Every FSM state is represented by one or more CPU states
- For example, reading the PIN requires multiple CPU states

→ Keyboard interrups, reading keys, storing text in memory, ...

- Not every CPU state is represented in the FSM

CPU states



3 cases for CPU states

- Sane state: A CPU state corresponding to an FSM state
- Transitory state: A CPU state during a transition, leading to a sane state
- Weird state: A CPU state which does not correspond to an FSM state

Example continued: A Simple Password Manager

```c
int main() {
 uint32_t pin, correct = 0;
 while(1) {
  pin = readPIN();
  if(pin * 2654435761u == 324783883u)
      correct = 1;

  if(correct) {
   showPasswords();
   break;
  } else printf("\nWrong PIN!\n");
 }
 return 0;
}
```

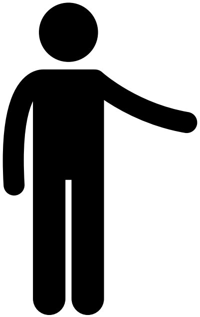
States

CPU State: TransitorySane

State: -Read PINcorrect?Show Password List

```c
uint32_t readPIN() {
 char buffer[16];
 printf("Enter PIN:\n");
 gets(buffer);
 if(getenv("DEBUG")) printf(buffer);
 return atoi(buffer);
}
void showPasswords() {
 FILE* stream;
 char* l = NULL;
 size_t len;
 stream = fopen("passwords", "r");
 if (stream == NULL) return;

 while(getline(&l, &len, stream) != -1)
   puts(l);
 free(l);
 fclose(stream);
}
```

The Weird State

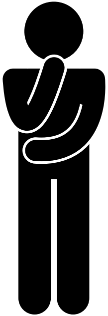


- CPU emulates the FSM

$\rightarrow$ Should only be in sane or tranistory state

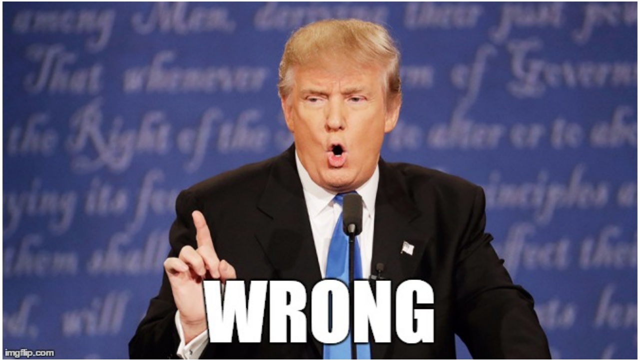
- How can the CPU enter the weird state?
  - Programming mistakes
  - Broken hardware (e.g., bit flips in memory)
  - Hardware bugs (e.g., CPU bugs)
  - ...
- Program does not know it is in weird state

Running in the Weird State

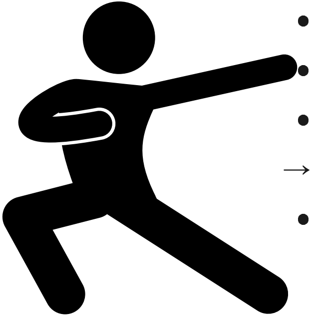


- Program continues executing
- Transitions might still be applied → on a weird state instead of a sane state
- Usually transforms one weird state into another weird state
- Weird machine, with many weird states
- We can “program” the weird machine to do something different than the original FSM

- Write program using code → translated into instructions executed by the CPU
- To program a device we have to generate instructions



Instructions as Program



- Get rid of the mindset that we require code for programming
- Applications accept input
- Does different things depending on input

→ Input programs the application

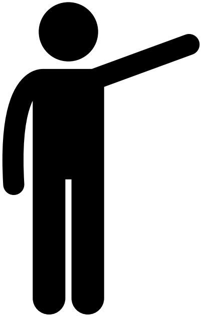
- Fine if input only leads from one sane state to another sane state

Instructions as Program in Weird States

- If application is in weird state and programmed using input...
- ...the attacker is controlling your computer

- An abstract definition of exploitation

Exploitation



Exploitation: Process starting in a sane state of an FSM

1. Setup: choose the right sane state which “allows” to get to a weird state
2. Instantiation: transition from sane state to weird state
3. Programming: program the weird machine

with the goal to break the security properties of the FSM

Back to the Example: A Simple Password Manager

- We want to enter a weird state
- Can we find a bug in the program?
- Can we abuse it to enter a weird state?
- First hint of a bug when compiling:

```
pwdman.c:(.text+0x2e): warning: the 'gets' function is dangerous
and should not be used.
```

→ Check the `man` page of `gets`

NAME

gets - get a string from standard input (DEPRECATED)

SYNOPSIS

```
#include <stdio.h>

char *gets(char *s);
```

DESCRIPTION

Never use this function.

gets() reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or **EOF**, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

RETURN VALUE

gets() returns **s** on success, and NULL on error or when end of file occurs while no characters have been read. However, given the lack of buffer overrun checking, there can be no guarantees that the function will even return.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
gets()	Thread safety	MT-Safe

CONFORMING TO

C89, C99, POSIX.1-2001.

LSB deprecates **gets**(). POSIX.1-2008 marks **gets**() obsolescent. ISO C11 removes the specification of **gets**() from the C language, and since version 2.16, glibc header files don't expose the function declaration if the **_ISOC11_SOURCE** feature test macro is defined.

BUGS

Never use **gets**(). Because it is impossible to tell without knowing the data in advance how many characters **gets**() will read, and because **gets**() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use **fgets**() instead.

Where is the Bug?

- Code part where `gets` is used:

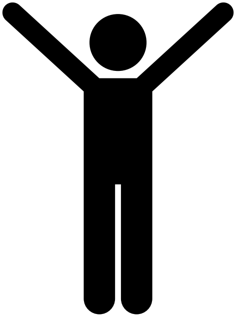
```
uint32_t readPIN() {
 char buffer[16];
 printf("Enter PIN:\n");
 gets(buffer);
 if(getenv("DEBUG")) printf(buffer);
 return atoi(buffer);
}
```

- The `buffer` array has space for 16 characters
- `gets` reads until `EOF`...

Trigger the Bug

```
% ./pwdman
Enter PIN:
1234

Wrong PIN!
Enter PIN:
0123456789012345678901234567890123456789
[1]    7106 segmentation fault (core dumped)  ./pwdman
pwdman[7486]: segfault at 31303938 ip 0000000031303938
              sp 00000000ffffcdc0 error 14 in
              libc-2.23.so[f7de2000+1b0000]
```

We are in a Weird State!

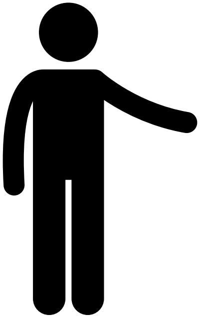


- We crash the program
- Crashing $\rightarrow$ not a state in our FSM

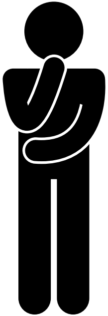
$\rightarrow$ Weird state due to a programming mistake

- #1: Why did we get into this weird state?
- #2: What is this weird state?
- #3: How can we program our weird machine to do something useful (instead of crashing)?

#1: The Why



- `gets` reads from the user until `EOF`
- Everything read is stored in an array
- Arrays have a defined size
- What if we write more data into the array?
- We write into something else adjacent in memory



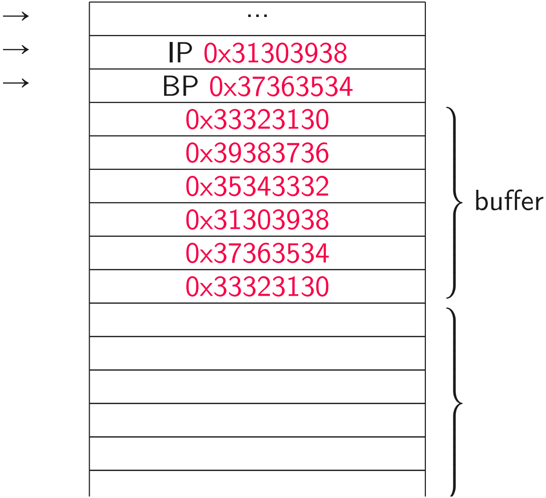
- What is next to the variable?
- It is a local variable, therefore it is on the stack
- Other local variables adjacent (none here)
- What else is on the stack?

#1: The Why - Recap: Stack

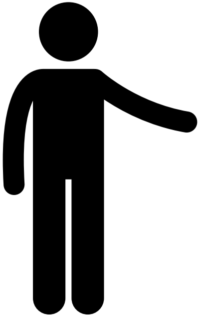
0x7FF...	saved return address	last frame
	saved base pointer	
	local variables	
	saved return address	
	saved base pointer	
	local variables	
0x000...	...	

#1: The Way - Overwriting the Stack

```c
uint32_t readPIN() {
  char buffer[16];
  printf("Enter PIN:\n");
  gets(buffer);
  if(getenv("DEBUG")) printf(
       buffer);
  return atoi(buffer);
}
```

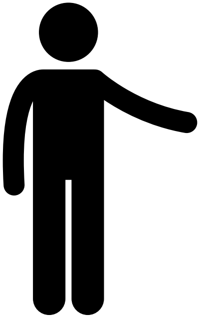


#2: The What



- We are somewhere (more specific: at address `0x31303938`)
- CPU tries to execute code at this address
- Probably nothing mapped at this address → pagefault
- Operating system kills application with a segmentation fault
- Weird state: CPU trying to execute code at an invalid address

#3: The How



- Bring the CPU in weird state by entering too many characters
- Control what the CPU executes by setting the instruction pointer
- We want to either
  - stay in a weird, but useful state, or
  - go to a (useful) sane state again
- Let's try to get to the sane state "Show Password List" first...

#3: The How - Go to an useful sane state

- We can let the CPU execute code at an arbitrary location
- The `showPasswords` function is at some location

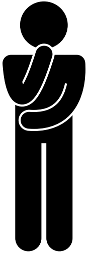
```
% readelf -s pwdman | grep showPasswords
64: 08048604   121 FUNC    GLOBAL DEFAULT   14 showPasswords
```

- PIN should look like this: `<padding>\x04\x86\x04\x08`
- `padding` fills the buffer (plus saved base pointer), address overwrites the saved instruction pointer

#3: The How - Break the Security Properties

```
echo "AAAAAAAAAAAAAAAAAAAAAAAAAAAA\x04\x86\x04\x08" | ./pwdman
Enter PIN:
root:toor

user:password1234

[1] 17074 segmentation fault (core dumped)  ./pwdman
```

- We broke the security properties of the FSM
- **Setup**: We started in the sane state “Read PIN”
- **Instantiation**: Too many characters led to a weird state
- **Programming**: We “programmed” the weird state using the input to move to the sane state “Show Password List”
- We have successfully developed an exploit

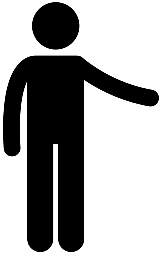


- Spatial memory safety violation to overwrite data

$\rightarrow$ Weird state

- Do we have to overwrite the saved instruction pointer?
- Other memory safety violations?
- Write in a more powerful “weird machine language”?

Do we have to overwrite the Instruction Pointer?

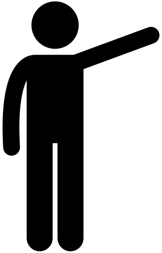


- No → just one "trick" to get into weird state
- Controlling the control flow → weird state
- More ways to change instruction pointer

→ function pointers, vtables, ...

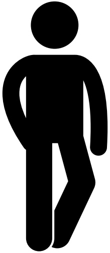
- Controlling the instruction pointer is not a requirement
- Control-flow hijacking is a "category of tricks"

So, there is an alternative?



- Got rid of the mindset that we require code to program
- Input as a way of programming a device
- Modify data used in an FSM state (transition)
- Changing data to something not intended in the original FSM
  $\longrightarrow$ weird state
- Assume `gets` bug is fixed, e.g., replaced by `fgets`

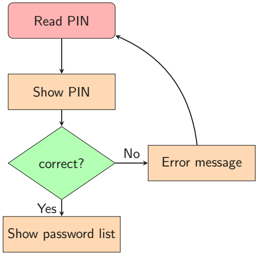
The fixed Code



```c
uint32_t readPIN() {
    char buffer[16];
    printf("Enter PIN:\n");
    fgets(buffer, 16, stdin);
    if(getenv("DEBUG")) printf(buffer);
    return atoi(buffer);
}
```

An Example (still continued): Simple Password Manager

```
uint32_t readPIN() {
    char buffer[16];
    printf("Enter PIN:\n");
    fgets(buffer, 16, stdin);
    if(getenv("DEBUG")) printf(buffer);
    return atoi(buffer);
}
```



- We ignored the "debug mode" before...
- One additional state in the FSM → echos the input
- Security property stays the same
- *It should be practically infeasible for an attacker to get the password list (output) if he does not know the PIN (input)*

Another Compiler Warning with `-Wformat-security`

- Compile with all warnings enabled (`-Wextra`)
- Still a warning

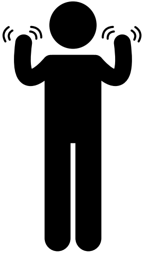
```
pwdman1.c:9:32: warning: format not a string literal and
                no format arguments [-Wformat-security]
  if(getenv("DEBUG")) printf(buffer);
                             ^
```

- What does the `man` page of `printf` say?

man 3 printf

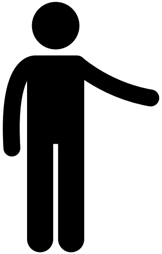
Code such as `printf(foo);` often indicates a bug, since `foo` may contain a `%` character. If `foo` comes from untrusted user input, it may contain `%n`, causing the `printf()` call to write to memory and creating a security hole.

Wait, what?



- `printf` can create a security hole?
- Why can `printf` write to memory?
- It is supposed to print text to the standard output...

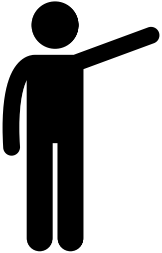
Re-cap: Format Strings



- We remember how to use `printf`:

```
printf("%d = 0x%x\n", 20, 20);
```

- Format string parameters (`%d`, `%s`, ...) convert function parameters to strings
- What if the number of format string parameters does not match the number of arguments?
- The function does not know
- Fetched form registers (first) and stack (afterwards)

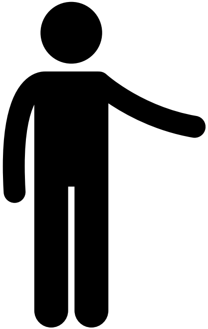


- `printf(user_input);` → user input is format string
- No parameters to the function
- Input does not contain a format string parameter → fine
- Format string parameter in the input → output a register value or stack value

Trigger the Bug

```
% DEBUG=1 ./pwdman1
Enter PIN:
%x %x %x %x
10 f76b55a0 f76f5858 25207825

Wrong PIN!
Enter PIN:
```

- Weird state - printing values from memory is not in our FSM
- How can we “program” this weird state?



- A little-known format string parameter: **%n**

man 3 printf

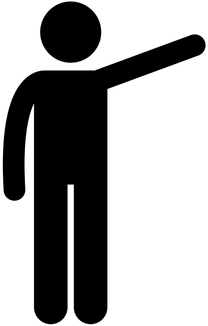
n The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an int *, or variant whose size matches the (optionally) supplied integer length modifier.

- Example:

```
int count;
printf("Some string %n\n", &count);
printf("Wrote %d charachters\n", count);
```

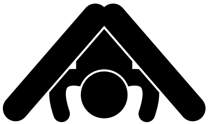
Prints `Wrote 12 characters`

Format Strings - Data Manipulation



- If there is an address on the stack, we can write to it
- Format string is on the stack → we can put any value onto the stack
- Can be the address to write to

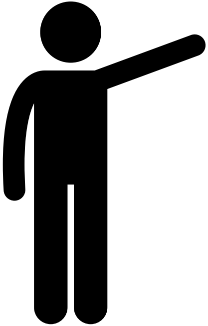
Playing around...



```
% echo "\x01\x02\x03\x04%x %x %x %x" |  \
    DEBUG=1 ./pwdman1
Enter PIN:
10 f7f945a0 f7fd4858 4030201
Wrong PIN!
Enter PIN:
```

```
% echo "\xb8\xcd\xff\xff%x %x %x %x" |  \
    DEBUG=1 ./pwdman1
Enter PIN:
����10 f7f945a0 f7fd4858 ffffcdb8
Wrong PIN!
Enter PIN:
```

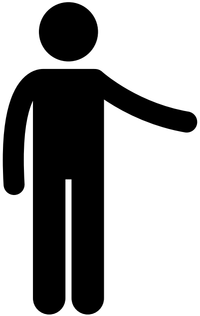
Programming the Weird State



```
% echo "\xb8\xcd\xff\xff%x %x %x %n" |  \
     DEBUG=1 ./pwdman1
Enter PIN:
����10 f7f945a0 f7fd4858 root:toor

user:password1234
```

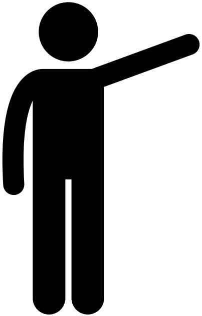
- With `%n`, we overwrote the `correct` variable at address `0xffffcdb8`
- Programmed the weird machine using the input...
- ...to transition to sane state "Show Password List"

More Memory Safety Violations



- There are many different memory safety violations
- All of them can get us into a weird state
- We have only seen 2 of them, but there are a lot more
- Memory safety violations are a "bag of tricks" from which we can take one to get into a weird state

More Powerful “Weird Programs”



- Our “weird machine programs” were quite simple

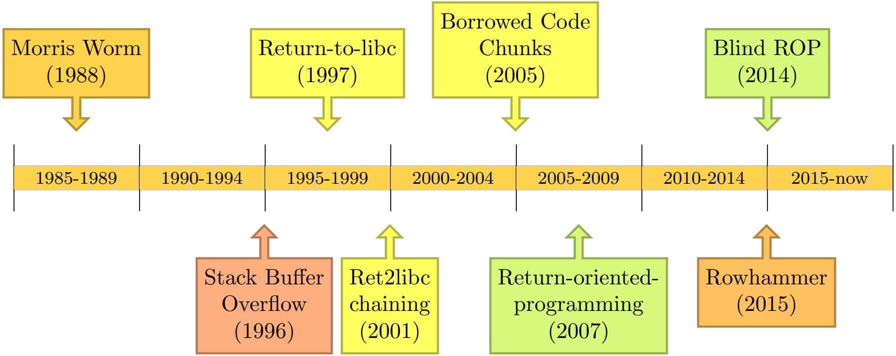
→ Jumped to a sane state of the FSM

- Instead
  - Inject own code and jump to that
  - Jump into the middle of a sane state
  - ...

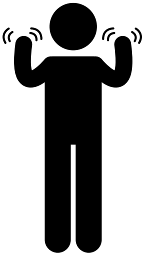
“Programming Languages” for the Weird Machine

For three decades

- people came up with tricks to get into weird states,
- and “programming languages” to program weird machines

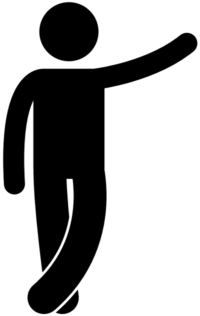


That sounds interesting, I want to learn more!



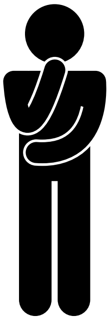
- There are many techniques and cool tricks
- Did not look at them → more important to understand concept
- Theory might be boring but helps understanding the techniques
- Participate in a CTF and try it yourself

Fix all the things



- We got rid of `gets`
- We got rid of the format-string vulnerability
- We could not find any other bugs
- The FSM emulator (= our code) looks secure

Non-exploitable Code?

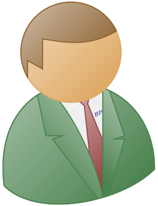


- Can we show that our code is now not exploitable?
- Not really → check all weird states whether they are exploitable
- How to know which weird states are reachable?
- Depends on the attacker model → what can an attacker do?
- Hard to think of attacker models not yet discovered

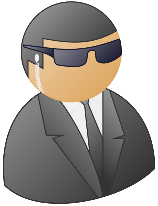
Exploitation

- Who is interested in exploitation?

Criminals



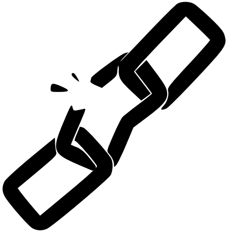
Vendors



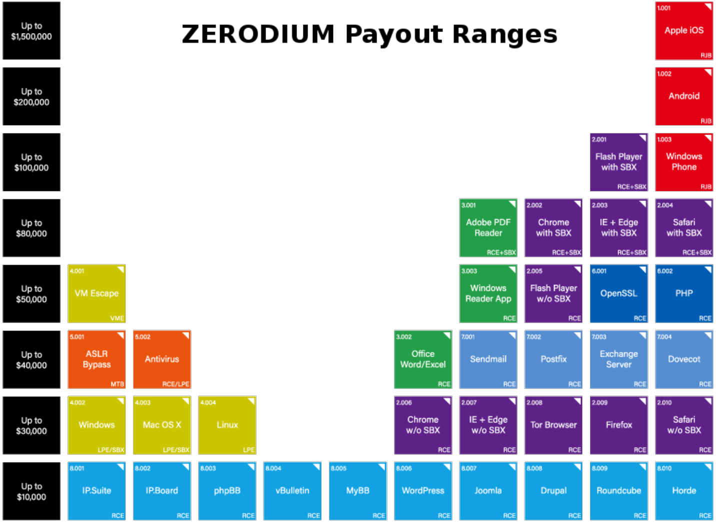
Governments

Zero-Days for Piracy

- Jailbreaks (e.g., getting root) on various devices:
  - iOS (multiple exploits)
  - Wii (buffer overflow in *The Legend of Zelda: Twilight Princess*).
  - PS2 (buffer overflow in the BIOS)
  - PS3 (heap overflow)
  - Xbox (buffer overflow in savegames)



Bug Bounty Programs



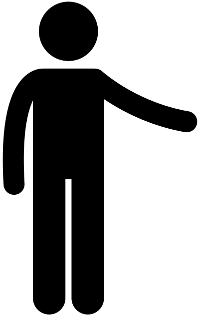
Zero-Days in Government

- Computer and network surveillance
- Sometimes use state-sponsored trojan horses (govware)

- Bundestrojaner (Germany)
- MiniPanzer and MegaPanzer (Switzerland)
- “Sicherheitspaket” (Austria)
- NSA Exploits (Shadow Broker Leak)

Blue Team aka Defenses

We want to defend against attacks

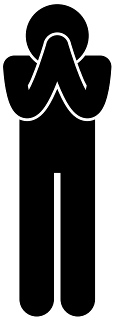


- Defense in CS is surprisingly hard
- In "classical war games", there is the 3:1 rule

→ An attacker needs 3 times as many soldiers as the defender

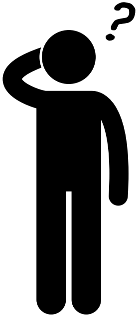
- Not a law (there are many exceptions) but rule of thumb

The defender has a disadvantage



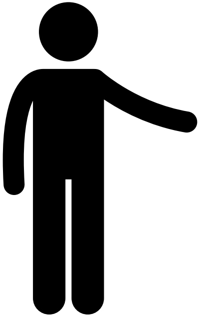
- In CS, the defender has a disadvantage
- Attacker: find one vulnerability
- Defender: protect against all possible attacks
- If the defender misses one vulnerability, the attacker wins
- "The best defense is a good offense" does not work

What do we do in CS?

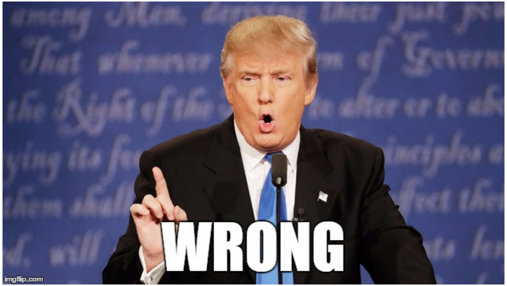


- Mainly two strategies
- Strategy #1: Red Team finds all bugs → Blue Team fixes them
- Strategy #2: Find generic mechanisms → Red Team cannot exploit the program

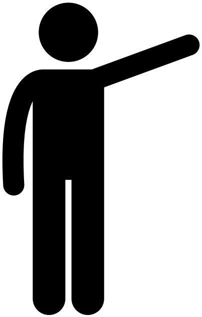
Strategy #1: Exploit. Fix. Feel Safe. Repeat



- Often, Strategy #1 is used $\rightarrow$ seems simple (and cheap)
- If a bug is discovered, fix it, done
- "It took an attacker/researcher more than n months to find a bug, so the cost of finding the next bug is $\geqslant n$ months"

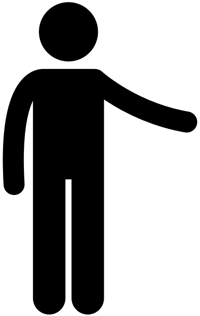


Re-cap: Weird machines



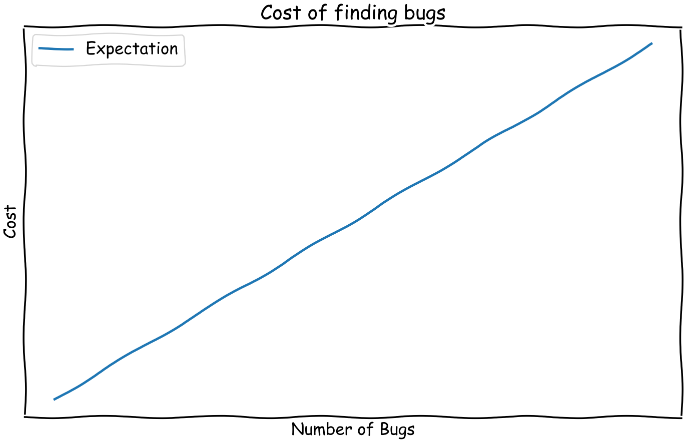
- We defined exploitation as a three-step procedure
  1. Setup: choose sane state which "allows" getting to a weird state
  2. Instantiation: transition from sane state to weird state
  3. Programming: program the weird machine
- The fix prevents one weird machine (or its "program")
- Similar bugs → similar weird machines

Strategy #1: Exploit. Fix. Feel Safe. Repeat

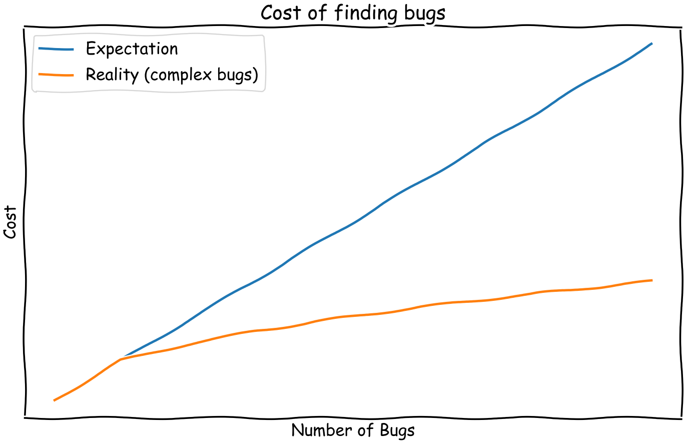


- If an attacker found one bug, there might be other similar bugs
- A lot easier to find and exploit similar bugs
- True until there are no similar bugs anymore

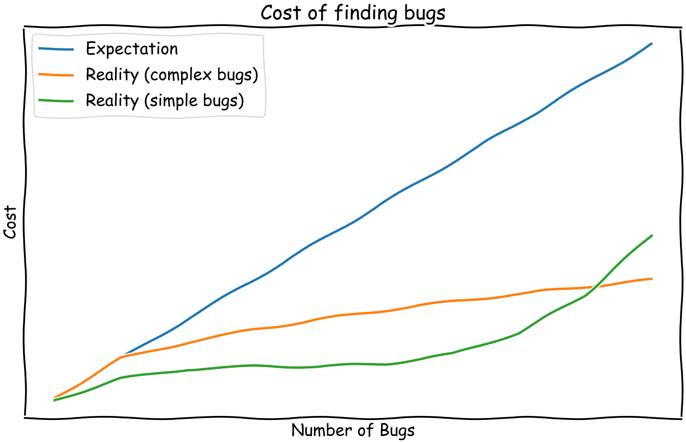
Strategy #1: Exploit. Fix. Feel Safe. Repeat



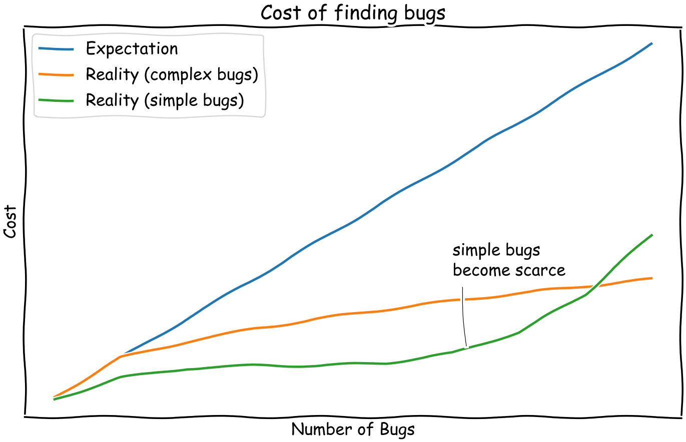
Strategy #1: Exploit. Fix. Feel Safe. Repeat



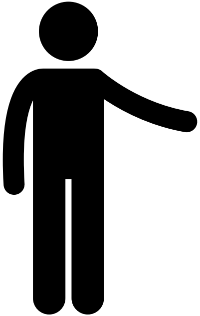
Strategy #1: Exploit. Fix. Feel Safe. Repeat



Strategy #1: Exploit. Fix. Feel Safe. Repeat

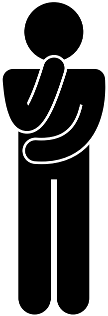


Strategy #2: The Academic Way



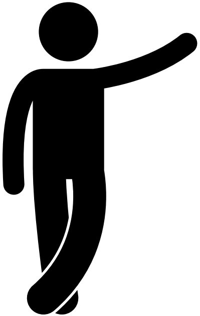
- Better: defense killing whole class of bugs, e.g. buffer overflows
- Can be extremely hard → not easy to find bug-free programs
- We already win if we prevent exploitation
- And we have a solid definition of exploitation

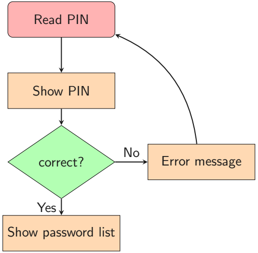
Strategy #2: The Academic Way



- Prevent one step of exploitation
- Cannot prevent Setup step → every transition is sane and the state is defined
- Try to prevent Instantiation and Programming step
- Start with Instantiation step
- We again use the Simple Password Manager as an example

An Example



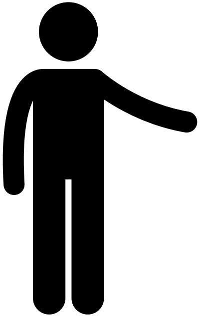



```
uint32_t readPIN() {
 char buffer[16];
 printf("Enter PIN:\n");
 gets(buffer);
 if(getenv("DEBUG"))
   printf(buffer);
 return atoi(buffer);
}
```

An Example

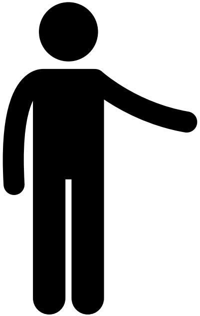
- We assume that the Red Team did not find the bugs (yet)
- We don't know about the `gets` and `printf` bug
- The problem the Blue Team has when defending:
  - The Blue Team has to roughly know about possible attacks
  - Protecting against a (yet) unknown attack is often not possible or comes with great costs (e.g. performance overhead)
- Assume we know about stack-buffer overflows

An Example



- Want to prevent Instantiation step
- Attacker should not get into weird state using a buffer overflow
- Program should rather die than being attacker controlled
- Remember: Stack overflow → overwrite the saved return address
- Cannot make it readonly (write permissions have page-level granularity)

An Example



- Simple idea: put a known (random) value between the buffer and the saved return address
- We call this value canary (yes, like the yellow bird)
- Canary is overwritten first
- On return, check whether the canary has the correct value
- If not → buffer overflow, kill program

Overwriting the Stack (Canary)

```c
uint32_t readPIN() {
 char buffer[16];
 printf("Enter PIN:\n");
 gets(buffer);
 if(getenv("DEBUG")) printf(buffer);
 return atoi(buffer);
}
```

→	...	
→	IP 0x35343332	
	BP 0x31303938	
→	0x01002236 0x37363534	} Canary
	0x33323130	buffer
	0x39383736	
	0x35343332	
	0x31303938	
	0x37363534	
	0x33323130	

Before return, check
canary → 0x01002236 ≠
0x37363534 → exit

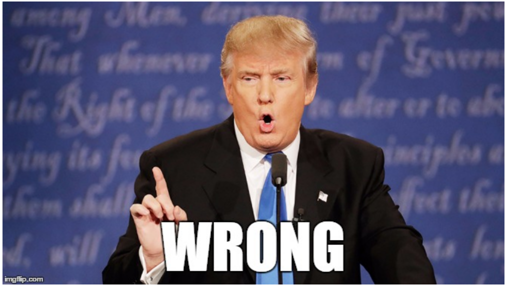
Trigger the Bug with Stack Canary

- Stack canaries are default in gcc
- However, only buffers larger than 8 bytes are protected
- We can use `-fstack-protector-all` to protect all buffers

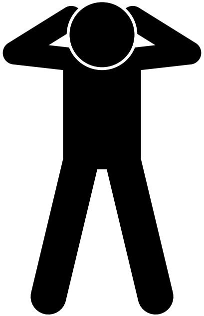
```
% gcc pwdman.c  -fstack-protector-all  -o pwdman
% ./pwdman
Enter PIN:
012345678901234567890123456789
*** stack smashing detected ***: ./pwdman terminated
[1]    7569 abort (core dumped)   ./pwdman
```

We fixed a class of bugs

- We fixed the class of stack-overflow bugs
- The canary protects every stack buffer from being used to get into a "weird state"



We fixed a class of bugs

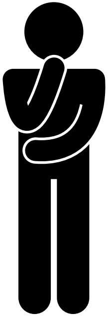


- Simple stack-buffer overflow cannot get into an exploitable weird state
- Leak canary using a different trick (e.g., `printf` bug, or out-of-bounds read)

→ Only prevented a part of a class of bugs

- Still other ways to get into a weird state
- We want something more generic, even if less powerful

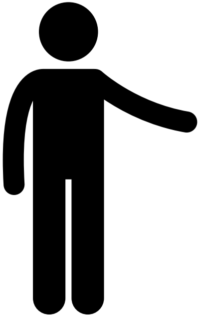
Any alternatives?



- Alternative to prevent the Instantiation step?
- Overwritting saved instruction pointer on the stack $\longrightarrow$ weird state

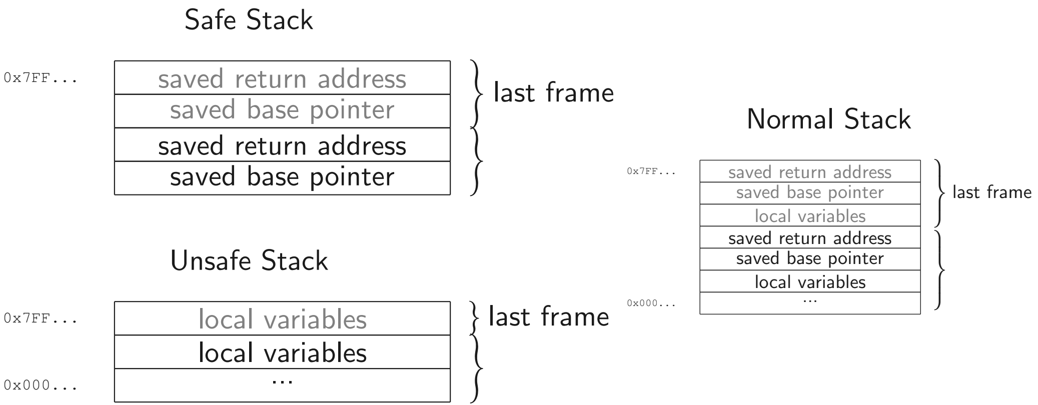
$\longrightarrow$ Separate saved return addresses and buffers

An alternative: Safe Stack



- Simple idea: two different stacks, a safe stack and an unsafe stack
- Simple variables and return values on the safe stack
- Buffers on the unsafe stack
- Buffer overflows cannot overwrite the return address anymore

Safe Stack

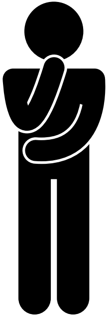

Safe Stack
0x7FF...
saved return address
saved base pointer
saved return address
saved base pointer
last frame
Unsafe Stack
0x7FF...
0x000...
local variables
local variables
...
last frame
Normal Stack
0x7FF...
0x000...
saved return address
saved base pointer
local variables
saved return address
saved base pointer
local variables
...
last frame

Trigger the Bug with Safe Stack

- clang supports safe stacks with a compile flag (not yet implemented in gcc)

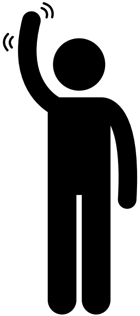
```
% clang pwdman.c -fsanitize=safe-stack -o pwdman
% ./pwdman
Enter PIN:
1234

Wrong PIN!
Enter PIN:
0123456789012345678901234567890123456789

Wrong PIN!
Enter PIN:
```

Think bigger!



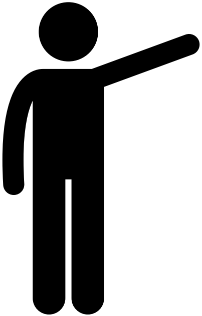
- Until now, we only prevented a small class of bugs
- It looks like a cat-and-mouse game
- It works and adds protection, but we have to combine a lot of countermeasures if we continue that way
- Every countermeasures costs (performance, memory, ...)
- We want something more generic, even if it is not as powerful as specific countermeasures

It's all about randomness

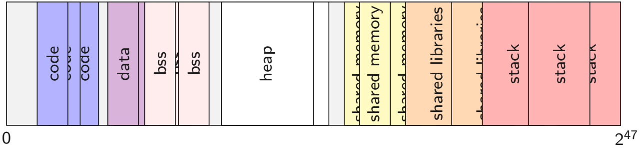


- Randomness is often used in security → probabilistic approach
- Assumption: attacker can jump to any memory location
- What if all memory locations are unpredictable?
- Attacker cannot reliably jump to a specific location anymore

Address Space Layout Randomization (ASLR)

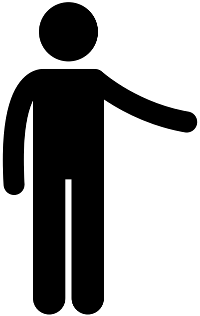


- Address Space Layout Randomization (ASLR) randomizes the position of program parts



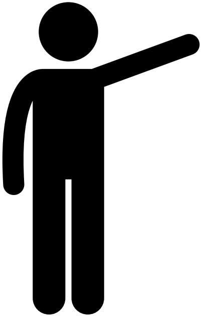
- Attacker cannot predict the location of a sane or injected state
- Powerful on 64-bit systems → huge address space (128 TB)

ASLR and its benefits



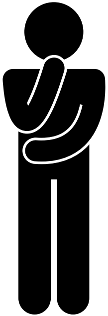
- ASLR is only a probabilistic countermeasure relying on two assumptions
  - No leak of addresses → breaks ASLR immediately
  - Randomization range is large enough → brute force breaks ASLR
- On 64-bit systems, ASLR makes exploitation really hard
- Advantage of ASLR: it costs nearly nothing → widespread use

ASLR in the real world



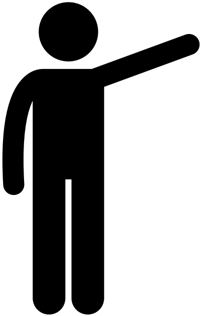
- As ASLR is a cheap but nevertheless effective countermeasure, it is widely used
  - Linux since 2005 (since 2014 in the kernel)
  - Windows since 2007
  - Android and iOS since 2011
  - Mac OS since 2011 (since 2012 in the kernel)
- Prevented many single bug exploits, as they fail with a high probability

Preventing the Programming step



- Assumption: attacker still found a way to get into a weird state
- Last ressort to prevent exploitation → make the Programming step infeasible
- Attacker uses the input stream to program the weird machine
- We could filter the input stream – but this is not always possible

Self awareness

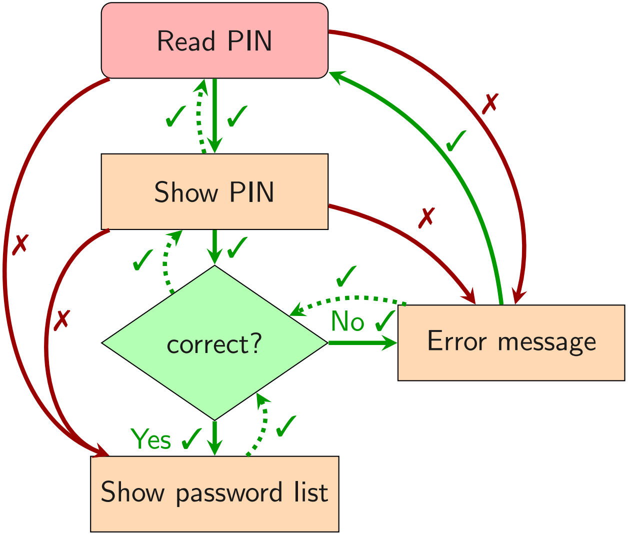


- Idea: make the FSM aware of itself!
- The FSM should know which states and transitions are allowed

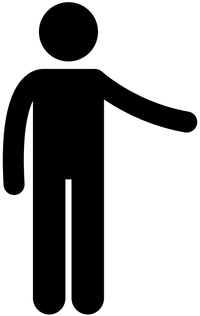
→ Prevent all transitions which are not in the original FSM

- Every state has to check whether
  - target of an indirect jump is correct according to the FSM
  - saved return address points to a previous state
- Forces the program to stay inside the FSM

Allowed and Disallowed transitions



Control-flow integrity



- Control-flow integrity sounds simple → difficult to implement
  - Control-flow graph must be correctly constructed
  - Function pointers cannot be protected if destination set is large
  - Some functions (e.g., library functions) have many call locations and therefore return locations
- Still, usable implementations in clang and from Microsoft
- Exploitation is still possible → integrity checks are often coarse-grained

Control-flow integrity - Example

```c
typedef void (*function)();
void help() {
 printf("Display this help message\
     n");
}
void unlock() {
 unlockPasswordManager();
}
void quit() {
 printf("Bye!\n");
 exit(0);
}
void usage() {
 printf("Usage: pwdman-ui <0-2>\n")
     ;
}
```

```c
void debug() {
 printf("Here is your shell\n");
 system("/bin/bash");
}
int main(int argc, char* argv[]) {
 function commands[] = {
   help, unlock, quit
 };
 function debugging[] = {
   debug
 };
 if(argc > 1) {
  commands[atoi(argv[1])]();
 } else usage();
}
```

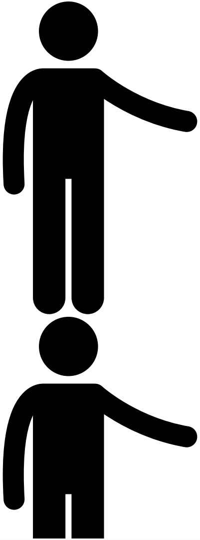
Control-flow integrity - Example

```
% clang pwdman-ui.c -o pwdman-ui
% ./pwdman-ui
Usage: pwdman-ui <0-2>
% ./pwdman-ui 0
Display this help message
% ./pwdman-ui 1
Enter PIN: ^C
% ./pwdman-ui 2
Bye!
% ./pwdman-ui 10
[1]    20659 segmentation fault (core dumped)  ./pwdman-ui 10
% ./pwdman-ui -1
Here is your shell
#
```

Control-flow integrity - Example

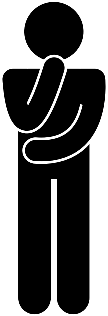
```
% clang pwdman-ui.c -fsanitize=cfi -flto -fvisibility=hidden \
    -fno-sanitize-trap=all -o pwdman-ui
% ./pwdman-ui
Usage: pwdman-ui <0-2>
% ./pwdman-ui 0
Display this help message
% ./pwdman-ui 10
pwdman-ui.c:43:9: runtime error: control flow integrity check
for type 'void ()' failed during indirect function call
0x2079616c70736944: note: (unknown) defined here
% ./pwdman-ui -1
pwdman-ui.c:43:9: runtime error: control flow integrity check
for type 'void ()' failed during indirect function call
0x000000293028: note: (unknown) defined here
```

Is that all we can do?



- We discussed techniques to prevent the Instantiation step
  - Canary
  - ASLR
- And control-flow integrity to prevent Programming step
- They provide good protection but can be circumvented
- Why use the countermeasures if they can be circumvented?

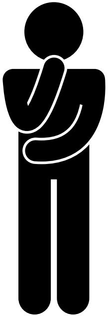
Costs and Raising the Bar



- Often arguments such as
  - "We have to increase the costs/raise the bar for an attacker"
  - "Many layers of security make it a lot harder for an attacker"
- That is partly true, however...
- ...in most cases there is a trade-off
- Increased cost for the attacker usually comes with increased cost for the user as well

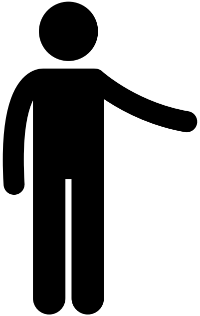
→ slower programs, increased memory consumption, ...

Costs and Raising the Bar



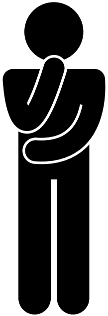
- User has to pay the costs all the time
- Attacker only has to pay them once
- A defender has to decide whether such a trade-off is worth for individual cases

Costs and Raising the Bar



- Presented countermeasures provide a good trade-off between cost and security
- This is one reason why they are widely used
- Future hardware might implement some countermeasures to reduce the costs
- What else can we do in the meantime?

Limit the damage



- Might not prevent attack from a sophisticated attacker

→ Restrict the attacker after the exploit

- Protect our system, even if application is controlled by the attacker

Sandboxing

Sandboxing - Demo

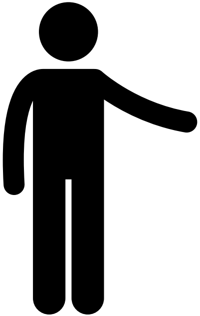
- Simple sandboxing with Docker can be as easy as running one command

```
% docker run --rm --read-only=true -i --cap-drop=all \
   --net=none -v $PWD:/app -t ubuntu /app/pwdman
Enter PIN: ��������
# ls
app  bin  boot dev  etc  home  lib  lib64  media  mnt
opt  proc  root  run  sbin  srv  sys  tmp  usr   var
# echo "test" > /tmp/test
sh: 4: cannot create /tmp/test: Read-only file system
# networkctl
IDX LINK             TYPE             OPERATIONAL SETUP
  1 lo               loopback         n/a         n/a

1 links listed.
```

Sandboxing - Demo

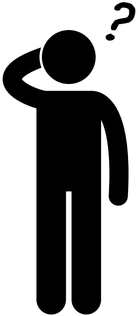
- An attacker cannot do much anymore
  - The file system is readonly, no files can be changed/created
  - No files of the host computer are visible, except the program and the password list
  - There is no network connection to easily exfiltrate data
- Even if our program is owned by an attacker, the attacker can at least not harm the rest of the system

Expect the worst

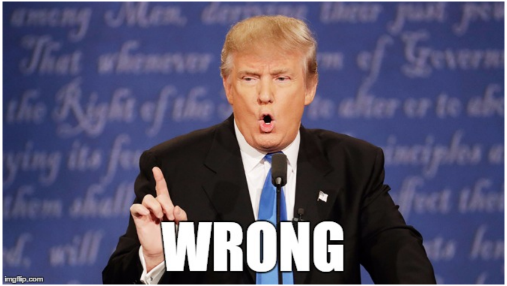


- Always expect the worst case that could happen!
- In this case: attacker found exploitable bug, circumvented all countermeasures, got a shell in the sandbox and was able to read the password file
- → No problem if file is encrypted, and key is derived from PIN
- (Assuming the crypto is good, and you used it correctly)

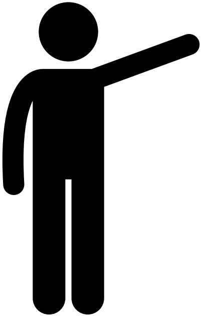
Why use a Sandbox then?



- If we encrypt the data, do we even benefit from a sandbox?
- Attacker cannot read the password file anyway

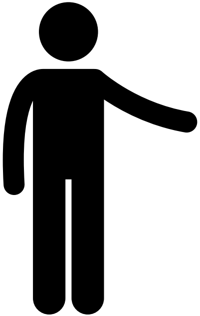


Always use a Sandbox!



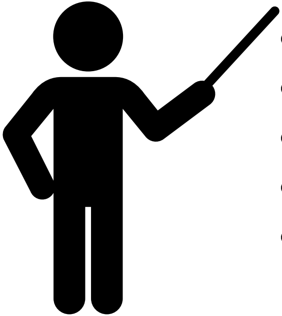
- Without sandbox, attacker can create/modify files
- Attacker could install a keylogger or other malicious software
- Or replace the password manager with a manipulated one leaking the PIN
- Best crypto does not help if system is compromised

Best practice



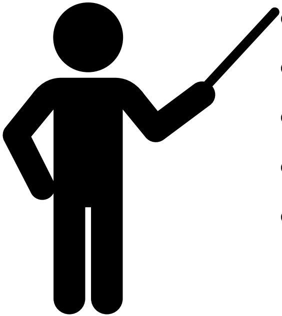
- Never assume perfect countermeasures or bug-free code
- Encrypt your data in case it leaks (it will at some point)
- Minimize privileges (e.g., a server should not run as root)
- Log everything – in case of an attack, you have a chance to find (and sue) the attacker
- Compiler can help to harden your application, e.g., using compile flags such as `-D_FORTIFY_SOURCE=2`

Take Aways



- Never ignore compiler warnings
- Don’t disable default counteremeasures (e.g., stack canaries)
- Enable countermeasures that are cheap, e.g., ASLR
- Consider stronger countermeasures, such as CFI
- Always consider sandboxing your application

Summary



- Defending software is hard, but not impossible
- Defenses are important to raise the cost for an attacker
- Security is a cat-and-mouse game full of repetitions
- The best countermeasure: don't have bugs in your code
- Realistic view: impossible to have bug free code, but try to reduce the number of bugs