A Brief Introduction to Memory Safety, Exploitation, and Countermeasures

Michael Schwarz
September 18, 2019

www.iaik.tugraz.at
Who is interested in exploitation?
Who is interested in exploitation?

- Criminals
Who is interested in exploitation?

- Criminals
- Vendors
• Who is interested in exploitation?

Criminals

Vendors

Governments
• 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)
• Jailbreaks (e.g., getting root) on various devices:
  • iOS (multiple exploits)
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)
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).
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)
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)
● Computer and network surveillance
Zero-Days in Government

- Computer and network surveillance
- Sometimes use state-sponsored trojan horses (govware)
Zero-Days in Government

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

- Bundestrojaner (Germany)
Zero-Days in Government

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

- Bundestrojaner (Germany)
- MiniPanzer and MegaPanzer (Switzerland)
Zero-Days in Government

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

- Bundestrojander (Germany)
- MiniPanzer and MegaPanzer (Switzerland)
- “Sicherheitspaket” (Austria)
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)
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
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
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
The complexer the programs, the more bugs

Source: http://www.cvedetails.com/vulnerabilities-by-types.php
There are two views on memory safety:

- Attackers try to violate memory safety
- Defenders try to ensure memory safety
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”
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
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
The Red Team are not (only) criminals, their work is essential for the 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.
• 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
• 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
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?
• 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 Achilless heel was his vulnerability in the Iliad, then Pariss poison tipped arrow was the exploit. ” (Kaspersky)
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 Achilless heel was his vulnerability in the Iliad, then Pariss poison tipped arrow was the exploit.” (Kaspersky)

→ Quite fuzzy
What is a “normal” program?²

• Programs: machines solving a certain problem(?)

²Most of the following ideas are from Halvar Flake / Thomas Dullien
What is a “normal” program?²

- Programs: machines solving a certain problem(?)
- Ideally, finite-state machines

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

\(^2\)Most of the following ideas are from Halvar Flake / Thomas Dullien
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

²Most of the following ideas are from Halvar Flake / Thomas Dullien
What is a “normal” program?

• Finite-state machines: states and transitions

Finite-state machines: states and transitions
What is a “normal” program?

- Finite-state machines: states and transitions
- Input: changes state to different state
What is a “normal” program?

- Finite-state machines: states and transitions
- Input: changes state to different state
- Finite-state machine (FSM) solves your problem
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
An Example: Simple Password Manager

- Security properties for your FSM
- Security properties based on inputs and outputs
• **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)*
• We have to write an emulator for our FSM
• We have to write an emulator for our FSM
• CPU has a lot more states than our FSM
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
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 interrupts, reading keys, storing text in memory, ...
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 interrupts, reading keys, storing text in memory, ...

Not every CPU state is represented in the FSM
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
3 cases for CPU states

- **Sane state**: A CPU state corresponding to an FSM state
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
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
```c
int main() {
    uint32_t pin, correct = 0;

    while(1) {
        pin = readPIN();
        if (pin * 2654435761u == 324783883u)
            correct = 1;

        if(correct) {
            showPasswords();
            break;
        } else
            printf("Wrong PIN!\n");
    }

    return 0;
}

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);
}
```

**States**

CPU State: Transitory

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

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

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);
}
```

**States**
**CPU State:** Transitory
**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("Wrong PIN!\n");
    }
    return 0;
}

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);
}
```

States
CPU State: Sane
State: Read PIN
int main() {
    uint32_t pin, correct = 0;
    while(1) {
        pin = readPIN();
        if(pin * 2654435761u == 324783883u)
            correct = 1;

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

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

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

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);
}
# 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("Wrong PIN!\n");
    }
    return 0;
}
```

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

```c
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);
}
```

**States**

**CPU State:** Transitory

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

        if (correct) {
            showPasswords();
            break;
        } else printf("Wrong PIN!\n");
    }

    return 0;
}

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);
    }

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

**States**

CPU State: Sane

State: correct?
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("Wrong PIN!\n");
            }
        }
    }
    return 0;
}
```

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

```c
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);
    }
}
```

**States**

**CPU State: Sane**

**State: correct?**
Example continued: A Simple Password Manager

```c
int main() {
    uint32_t pin, correct = 0;
    while (1) {
        pin = readPIN();
        if (pin * 265435761u == 324783883u)
            correct = 1;
        if (correct) {
            showPasswords();
            break;
        } else
            printf("Wrong PIN!\n");
    }
    return 0;
}

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);
}
```

**States**

**CPU State:** Sane

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

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

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);
}
```

**States**

**CPU State:** Transitory

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

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

States
CPU State: Sane
State: Show Password List

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

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

        if(correct) {
            showPasswords();
            break;
        } else printf("Wrong PIN!
");

    }
    return 0;
}

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);
}
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("Wrong PIN!\n");  
    
    return 0;  
}
```

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

```c
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);  
}
```

**States**

**CPU State:** Sane  
**State:** Show Password List
int main() {
    uint32_t pin, correct = 0;
    while(1) {
        pin = readPIN();
        if (pin * 2654435761u == 324783883u)
            correct = 1;

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

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

        if(correct) {
            showPasswords();
            break;
        } else
            printf("Wrong PIN!
");
    }
    return 0;
}

uint32_t readPIN() {
    char buffer[16];
    printf("Enter PIN:
");
    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);
}

States
CPU State: Sane
State: Show Password List
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("Wrong PIN!
");
    }
    return 0;
}
```

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

```c
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);
}
```

States
CPU State: Sane
State: Show Password List
int main() {
    uint32_t pin, correct = 0;
    while(1) {
        pin = readPIN();
        if(pin * 2654435761u == 324783883u)
            correct = 1;

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

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);
}
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("Wrong PIN!\n");
    }
    return 0;
}
```

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

```c
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);
}
```

**States**

CPU State: Sane

State: Show Password List
The Weird State

- CPU emulates the FSM
  - 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
The Weird State

- CPU emulates the FSM
  → Should only be in sane or tranistory state
- How can the CPU enter the weird state?
The Weird State

- CPU emulates the FSM
  → 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)
  - ...

Michael Schwarz — www.iaik.tugraz.at
The Weird State

- CPU emulates the FSM
  → Should only be in sane or transistor 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
• Program continues executing
Running in the Weird State

- Program continues executing
- Transitions might still be applied → on a weird state instead of a sane 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
• 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
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
• Write program using code → translated into instructions executed by the CPU
• To program a device we have to generate instructions
• Write program using code → translated into instructions executed by the CPU
• To program a device we have to generate instructions
• Get rid of the mindset that we require code for programming
• Get rid of the mindset that we require code for programming
• Applications accept input
• Get rid of the mindset that we require code for programming
• Applications accept input
• Does different things depending on input
• Get rid of the mindset that we require code for programming
• Applications accept input
• Does different things depending on input
→ Input programs the application
• 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
• If application is in weird state and programmed using input...
• If application is in weird state and programmed using input...
• ...the attacker is controlling your computer
• If application is in weird state and programmed using input...
• ...the attacker is controlling your computer

• An abstract definition of exploitation
Exploitation: Process starting in a sane state of an FSM
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
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
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
• We want to enter a **weird** state
• We want to enter a **weird** state
• Can we find a **bug** in the program?
• We want to enter a **weird** state
• Can we find a **bug** in the program?
• Can we abuse it to enter a weird state?
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.
```
• 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:

```plaintext
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).

<table>
<thead>
<tr>
<th>Interface</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>gets</td>
<td>Thread safety</td>
<td>MT-Safe</td>
</tr>
</tbody>
</table>

CONFORMING TO

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.

For more information, see CWE-242 (aka "Use of Inherently Dangerous Function") at http://cwe.mitre.org/data/definitions/242.html

SEE ALSO
read(2), write(2), ferror(3), fgetc(3), fgets(3), fgetwc(3), fgetws(3), fopen(3), fread(3), fseek(3), getline
• Code part where `gets` is used:

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

- Code part where `gets` is used:

```c
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
• Code part where `gets` is used:

```c
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`...
% ./pwdman
Enter PIN:
1234
Wrong PIN!
Enter PIN:
% ./pwdman
Enter PIN:
1234

Wrong PIN!
Enter PIN:
0123456789012345678901234567890123456789
% ./pwdman
Enter PIN:
1234

Wrong PIN!
Enter PIN:
0123456789012345678901234567890123456789
[1] 7106 segmentation fault (core dumped) ./pwdman
pwdman[7486]: segfault at 31303938 ip 0000000031303938
sp 00000000ffffffc0 error 14 in libc-2.23.so[f7de2000+1b0000]
We crash the program
We are in a Weird State!

- We crash the program
- Crashing → not a state in our FSM

Michael Schwarz — www.iaik.tugraz.at
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
We are in a Weird State!

- We crash the program
- Crashing → not a state in our FSM
  → Weird state due to a programming mistake
- #1: Why did we get into this weird state?
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?
We are in a Weird State!

- We crash the program
- Crashing $\rightarrow$ not a state in our FSM
- 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)?
• *gets* reads from the user until EOF
• gets reads from the user until EOF
• Everything read is stored in an array
• gets reads from the user until EOF
• Everything read is stored in an array
• Arrays have a defined size
• `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?
- 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?
What is next to the variable?

It is a local variable, therefore it is on the stack.
What is next to the variable?
- It is a local variable, therefore it is on the stack
- Other local variables adjacent (none here)
• 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
saved base pointer
local variables

0x000...
```

last frame
#1: The Why - Recap: Stack

0x7FF...
- saved return address
- saved base pointer
- local variables
- saved return address

\{ \text{last frame} \}

0x000...

\{ \text{current frame} \}
#1: The Why - Recap: Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7FF...</td>
<td>saved return address</td>
<td>last frame</td>
</tr>
<tr>
<td></td>
<td>saved base pointer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>local variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>saved return address</td>
<td></td>
</tr>
<tr>
<td></td>
<td>saved base pointer</td>
<td>current frame</td>
</tr>
<tr>
<td>0x000...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
#1: The Why - Recap: Stack

0x7FF...
- saved return address
- saved base pointer
- local variables

} \quad \text{last frame}

0x000...
- saved return address
- saved base pointer
- local variables

} \quad \text{current frame}

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

Return, continue at 0x31303938
• We are somewhere (more specific: at address 0x31303938)
• We are somewhere (more specific: at address 0x31303938)
• CPU tries to execute code at this address
• We are somewhere (more specific: at address 0x31303938)
• CPU tries to execute code at this address
• Probably nothing mapped at this address → pagefault
• 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
• 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
Bring the CPU in **weird state** by entering too many characters
Bring the CPU in **weird state** by entering too many characters

Control what the CPU executes by setting the **instruction pointer**
• 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**
• 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...
• We can let the CPU execute code at an arbitrary location
• 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
```
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.
echo "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x04\x86\x04\x08" | ./pwdman
echo "AAAAAAAAAAAAAAAAAAAAAAAAAAAAA\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
• We broke the security properties of the FSM
• **Setup**: We started in the sane state “Read PIN”
• 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**
Summary

- 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”
Summary

- 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
Can we do more?

- Spatial memory safety violation to overwrite data
  → Weird state
Can we do more?

- Spatial memory safety violation to overwrite data
  → Weird state
- Do we have to overwrite the saved instruction pointer?
Can we do more?

- Spatial memory safety violation to overwrite data
  → Weird state
- Do we have to overwrite the saved instruction pointer?
- Other memory safety violations?
Can we do more?

- **Spatial memory safety violation** to overwrite data
  → 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
Do we have to overwrite the Instruction Pointer?

- No → just one “trick” to get into weird state
- Controlling the control flow → weird state
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, ...
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
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
So, there is an alternative?

• Got rid of the mindset that we require code to program
• Input as a way of programming a device
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)
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 → weird state
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 → weird state
- Assume **gets** bug is fixed, e.g., replaced by **fgets**
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

```c
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...
An Example (still continued): Simple Password Manager

```c
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
An Example (still continued): Simple Password Manager

```c
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
An Example (still continued): Simple Password Manager

```c
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`)
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.
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?
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.
```
• `printf` can create a security hole?
• printf can create a security hole?
• Why can printf write to memory?
• `printf` can create a security hole?
• Why can `printf` write to memory?
• It is supposed to print text to the standard output...
We remember how to use `printf`:
```
printf("%d = 0x%x\n", 20, 20);
```
We remember how to use `printf`:

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

Format string parameters (`%d`, `%s`, ...) convert function parameters to strings.
We remember how to use `printf`:
```c
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?
- We remember how to use `printf`:
  ```c
  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
We remember how to use `printf`:
\[
\text{printf}("%d = 0x%x\n", 20, 20);
\]

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)
Re-cap: Format Strings

- `printf(user_input);` → user input is format string
Re-cap: Format Strings

- `printf(user_input);` → user input is format string
- No parameters to the function
Re-cap: Format Strings

- `printf(user_input);` → user input is format string
- No parameters to the function
- Input does not contain a format string parameter → fine
• `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
% DEBUG=1 ./pwdman1
Enter PIN:
%x %x %x %x
% DEBUG=1 ./pwdman1
Enter PIN:
%x %x %x %x
10 f76b55a0 f76f5858 25207825
Wrong PIN!
Enter PIN:
```
% 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
% 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

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

Prints

Wrote 12 characters
• 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.
• 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:

```c
int count;
printf("Some string %n\n", &count);
printf("Wrote %d characters\n", count);
```
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:**

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

**Prints** `Wrote 12 characters`
• If there is an address on the stack, we can write to it
• If there is an address on the stack, we can write to it
• Format string is on the stack $\rightarrow$ we can put any value onto the stack
• If there is an **address** on the stack, we can **write** to it
• **Format string** is on the stack $\rightarrow$ we can **put any value** onto the stack
• Can be the **address** to write to
% echo "\x01\x02\x03\x04%x %x %x %x" | \n  DEBUG=1 ./pwdman1
Playing around...

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

% echo "\x01\x02\x03\x04\x %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 %x" | \
  DEBUG=1 ./pwdman1
Playing around...

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

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

Enter PIN:
??10 f7f945a0 f7fd4858 root:toor

user:password1234
Programming the Weird State

% echo "\xb8\xcd\xff\xff\%x \%x \%x \%n" | \ 
   DEBUG=1 ./pwdman1

Enter PIN:

user: password1234

- With \%n, we overwrote the correct variable at address 0xffffffffdcdb8
% 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 0xffffffffcdb8

• Programmed the weird machine using the input...
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 0xffffffffc0db8
• Programmed the weird machine using the input...
• ...to transition to sane state “Show Password List”
• There are many different memory safety violations
• There are many different memory safety violations
• All of them can get us into a weird state
• 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
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.
• Our “weird machine programs” were quite simple
Our “weird machine programs” were quite simple

→ Jumped to a sane state of the FSM
• Our “weird machine programs” were quite simple
→ Jumped to a sane state of the FSM
• Instead
Our “weird machine programs” were quite simple
→ Jumped to a sane state of the FSM
• Instead
  • Inject own code and jump to that
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
• 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
  • ...

Michael Schwarz — www.iaik.tugraz.at
For three decades

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

```
<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1985-89</td>
<td>Morris Worm (1988)</td>
</tr>
<tr>
<td>1990-94</td>
<td>Stack Buffer Overflow (1996)</td>
</tr>
<tr>
<td>2010-14</td>
<td>Blind ROP (2014)</td>
</tr>
<tr>
<td>2015-Now</td>
<td>Rowhammer (2015)</td>
</tr>
</tbody>
</table>
```
That sounds interesting, I want to learn more!

- There are many techniques and cool tricks
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
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
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 CTF and try it yourself
• We got rid of `gets`
Fix all the things

- We got rid of `gets`
- We got rid of the format-string vulnerability
Fix all the things

• We got rid of `gets`
• We got rid of the format-string vulnerability
• We could not find any other bugs
• 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
• Can we show that our code is now not exploitable?
• Can we show that our code is now not exploitable?
• Not really → check all weird states whether they are exploitable
• 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?
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?
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
Blue Team aka Defenses
We want to defend against attacks

- Defense in CS is surprisingly hard
We want to defend against attacks

- Defense in CS is surprisingly hard
- In “classical war games”, there is the 3:1 rule
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
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
• In CS, the defender has a disadvantage
• Attacker: find one vulnerability
The defender has a disadvantage

- In CS, the defender has a disadvantage
- Attacker: find one vulnerability
- Defender: protect against all possible attacks
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 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
What do we do in CS?

- Mainly two strategies
- Strategy #1: Red Team finds all bugs → Blue Team fixes them
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
• Often, Strategy #1 is used → seems simple (and cheap)
Strategy #1: Exploit. Fix. Feel Safe. Repeat

- Often, Strategy #1 is used → seems simple (and cheap)
- If a bug is discovered, fix it, done
• Often, Strategy #1 is used → 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 $\geq n$ months”
Strategy #1: Exploit. Fix. Feel Safe. Repeat

- Often, Strategy #1 is used → 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 $\geq n$ months”
• We defined exploitation as a three-step procedure
  1. **Setup**: choose sane state which “allows” getting to a weird state
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
We defined exploitation as a three-step procedure:

1. **Setup**: choose a sane state which “allows” getting to a weird state.
2. **Instantiation**: transition from a sane state to a weird state.
3. **Programming**: program the weird machine.
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”)
We defined exploitation as a three-step procedure:
1. **Setup**: choose a sane state that "allows" getting to a weird state
2. **Instantiation**: transition from a sane state to a 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.
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
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

Cost of finding bugs

Cost

Number of Bugs

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

Cost of finding bugs

- Expectation
- Reality (complex bugs)

Number of Bugs

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

Cost of finding bugs

- Expectation
- Reality (complex bugs)
- Reality (simple bugs)

Cost

Number of Bugs
Strategy #1: Exploit. Fix. Feel Safe. Repeat

![Graph showing the cost of finding bugs over the number of bugs. The graph compares the expectation with reality for simple and complex bugs.]

- **Expectation**
- **Reality (complex bugs)**
- **Reality (simple bugs)**

**Cost of finding bugs**

- Simple bugs become scarce
Strategy #2: The Academic Way

- Better: defense killing whole class of bugs, e.g. buffer overflows
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
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
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
Strategy #2: The Academic Way

- Prevent one step of exploitation
- Cannot prevent Setup step → every transition is sane and the state is defined
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
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
• 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
```c
uint32_t readPIN() {
    char buffer[16];
    printf("Enter PIN:\n");
    gets(buffer);
    if(getenv("DEBUG"))
        printf(buffer);
    if (atoi(buffer))
        return atoi(buffer);
    return 0;
}
```
• We assume that the Red Team did not find the bugs (yet)
• We assume that the Red Team did not find the bugs (yet)
• We don’t know about the `gets` and `printf` bug
• 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
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)
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
- Want to prevent Instantiation step
- Attacker should not get into *weird state* using a buffer overflow
- Want to prevent Instantiation step
- Attacker should not get into weird state using a buffer overflow
- Program should rather die than being attacker controlled
- 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
• 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)
• Simple idea: put a known (random) value between the buffer and the saved return address
• 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)
• 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
• 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
• 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
uint32_t readPIN() {
    char buffer[16];
    printf("Enter PIN:\n");
    gets(buffer);
    if (getenv("DEBUG")) printf(buffer);
    return atoi(buffer);
}
uint32_t readPIN() {
    char buffer[16];
    printf("Enter PIN:\n");
    gets(buffer);
    if (getenv("DEBUG")) printf(buffer);
    return atoi(buffer);
}
uint32_t readPIN() {
  char buffer[16];
  printf("Enter PIN:\n");
  gets(buffer);
  if(getenv("DEBUG")) printf(buffer);
  return atoi(buffer);
}
uint32_t readPIN() {
    char buffer[16];
    printf("Enter PIN:\n");
    gets(buffer);
    if (getenv("DEBUG")) printf(buffer);
    return atoi(buffer);
}
uint32_t readPIN() {
    char buffer[16];
    printf("Enter PIN:\n");
    gets(buffer);
    if (getenv("DEBUG")) printf(buffer);
    return atoi(buffer);
}
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);
}
```

Before return, check canary → 0x01002236 ≠ 0x37363534 → exit
- Stack canaries are *default* in gcc
• Stack canaries are **default** in gcc
• However, **only** buffers **larger than 8 bytes** are protected
• 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

```bash
% gcc pwdman.c -fstack-protector-all -o pwdman
```
• 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
```
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

```bash
% 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
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

- 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
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)
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
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 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
• Randomness is often used in security $\rightarrow$ probabilistic approach
• Randomness is often used in security → probabilistic approach
• Assumption: attacker can jump to any memory location
- Randomness is often used in security → probabilistic approach
- Assumption: attacker can jump to any memory location
- What if all memory locations are unpredictable?
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) randomizes the position of program parts.

- Attacker cannot predict the location of a sane or injected state.
- Powerful on 64-bit systems with huge address space (128 TB).
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 with a huge address space (128 TB).
- Address Space Layout Randomization (ASLR) randomizes the position of program parts

![Diagram showing the randomization of ASLR across different memory sections: code, data, bss, heap, shared memory, shared libraries, stack. The diagram illustrates the random distribution of these sections across the address space.]

0 \rightarrow 2^{47}
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
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 and its benefits

• ASLR is only a probabilistic countermeasure relying on two assumptions
  • No leak of addresses → breaks ASLR immediately
ASLR and its benefits

- ASLR is only a probabilistic countermeasure relying on two assumptions
  - No leak of addresses $\rightarrow$ breaks ASLR immediately
  - Randomization range is large enough $\rightarrow$ brute force breaks ASLR
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**
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
• Assumption: attacker still found a way to get into a weird state
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
Preventing the Programming step

- Assumption: attacker still found a way to get into a weird state
- Last resort to prevent exploitation → make the Programming step infeasible
- Attacker uses the input stream to program the weird machine
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
• Idea: make the FSM aware of itself!
• Idea: make the FSM *aware of itself*!
• The FSM should know which states and transitions are allowed
• 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
• 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
• 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
• 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
• 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

[Diagram showing allowed and disallowed transitions involving Read PIN, Show PIN, correct?, Error message, and Show password list.]
Control-flow integrity sounds simple → difficult to implement
Control-flow integrity sounds simple → difficult to implement
  - Control-flow graph must be correctly constructed
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
• 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
- 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
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
Is that all we can do?

- We discussed techniques to prevent the Instantiation step.
• We discussed techniques to prevent the Instantiation step
  • Canary
  • ASLR
Is that all we can do?

- We discussed techniques to prevent the Instantiation step
  - Canary
  - ASLR
- And control-flow integrity to prevent Programming step
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
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?
Often arguments such as
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”
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...
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
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
• 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, ...
• User has to pay the costs all the time
User has to pay the costs all the time
Attacker only has to pay them once
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
• Presented countermeasures provide a good trade-off between cost and security
Presented countermeasures provide a good trade-off between cost and security.

This is one reason why they are widely used.
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.
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
Limit the damage

- Might not prevent attack from a sophisticated attacker
  → Restrict the attacker after the exploit
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
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:
• Simple sandboxing with Docker can be as easy as running one command

```bash
% docker run --rm --read-only=true -i --cap-drop=all \ 
--net=none -v $PWD:/app -t ubuntu /app/pwdman
Enter PIN: ❂ ❂ ❂ ❂ ❂ ❂ ❂ ❂ ❂ ❂
```

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.
```
Simple sandboxing with Docker can be as easy as running one command:

```bash
% 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
```
• An attacker cannot do much anymore
• 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
• 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
• Always expect the **worst case** that could happen!
• 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
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
• 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)
• If we encrypt the data, do we even benefit from a sandbox?
Why use a Sandbox then?

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

- Without sandbox, attacker can create/modify files
- Attacker could install a keylogger or other malicious software
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
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
• Never assume perfect countermeasures or bug-free code
• Never assume perfect countermeasures or bug-free code
• Encrypt your data in case it leaks (it will at some point)
• 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)
• 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
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 \(-D\_FORTIFY\_SOURCE=2\)
Never ignore compiler warnings
• Never ignore compiler **warnings**
• Don’t disable default counterememures (e.g., stack canaries)
Take Aways

- Never ignore compiler warnings
- Don’t disable default countermeasures (e.g., stack canaries)
- Enable countermeasures that are cheap, e.g., ASLR
Take Aways

- Never ignore compiler warnings
- Don’t disable default countermeasures (e.g., stack canaries)
- **Enable countermeasures** that are cheap, e.g., ASLR
- Consider stronger countermeasures, such as CFI
• Never ignore compiler warnings
• Don’t disable default countermeasures (e.g., stack canaries)
• Enable countermeasures that are cheap, e.g., ASLR
• Consider stronger countermeasures, such as CFI
• Always consider sandboxing your application
Defending software is hard, but not impossible
• Defending software is **hard**, but *not impossible*
• Defenses are important to raise the cost for an attacker
• 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
• 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
• 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
Any Questions?