Secure Software Development

Memory Corruption II & Environment

Daniel Gruss, Vedad Hadzic, Lukas Maar, Stefan Gast, Marcel Nageler

27.10.2023

Winter 2023/24, www.iaik.tugraz.at

Table of contents

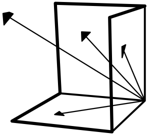
1. Use-after-free
2. Format Strings
3. Type Confusion
4. Environment Variables Problems
5. File System Pitfalls

PREVIOUSLY ON

SSD

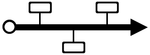
Memory Safety Violation

We can distinguish between two types of memory safety violation



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

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



Temporal violation: memory access refers to an invalid object

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

Memory Safety Violation Overview

Overflow (last lecture)

- Stack overflow
- Heap overflow
- Integer overflow

Invalid Memory (this lecture)

- Use-after-free
- Format string
- Type confusion

OVERFLOW

Overflow Summary

Overflows...

- are the most common forms of memory safety violation
- are mostly caused by missing bound checks
- can be abused to read from and write to memory
- might occur on buffers and integers
- exist in nearly every programming language (some exceptions)

INVALID MEMORY

Use-after-free

Use-after-free

- Referencing a resource after it was freed

Use-after-free

- Referencing a resource after it was freed
- C/C++ does not invalidate pointer when freeing its memory

Use-after-free

- Referencing a resource after it was freed
- C/C++ does not invalidate pointer when freeing its memory
- Such pointers are called dangling pointers

Use-after-free

- Referencing a resource after it was freed
- C/C++ does not invalidate pointer when freeing its memory
- Such pointers are called dangling pointers
- Also possible without dynamic memory (destroyed scope)

Use-after-free

Context 1 :

```
p = malloc(size) ;
// ...
free( p );
// ...
p = 0;
```

Use-after-free

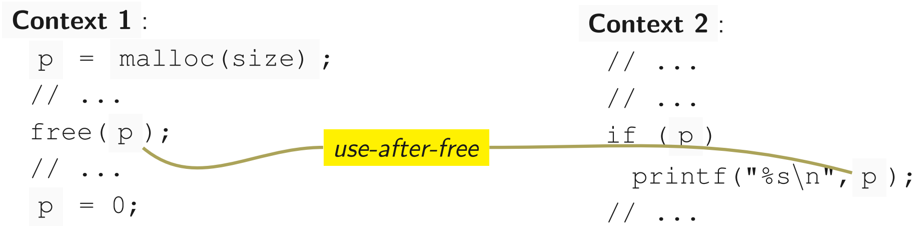
Context 1 :

```
p = malloc(size) ;
// ...
free( p );
// ...
p = 0;
```

Context 2 :

```
// ...
// ...
if ( p )
  printf("%s\n", p );
// ...
```

Use-after-free



Context 1 :

```
p = malloc(size) ;
// ...
free( p );
// ...
p = 0;
```

use-after-free

Context 2 :

```
// ...
// ...
if ( p )
  printf("%s\n", p );
// ...
```

Use-after-free

Resource

Context 1 :

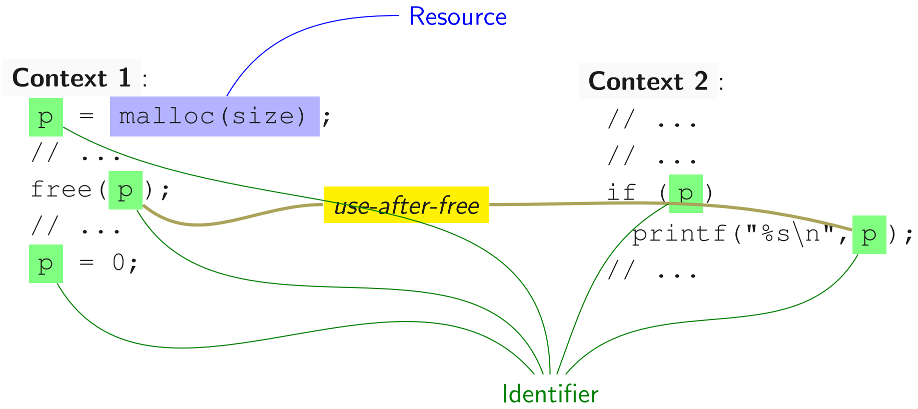
```
p = malloc(size);
// ...
free(p);
// ...
p = 0;
```

use-after-free

Context 2 :

```
// ...
// ...
if (p)
  printf("%s\n", p);
// ...
```

Use-after-free



Resource

Context 1:

```
p = malloc(size);
// ...
free(p);
// ...
p = 0;
```

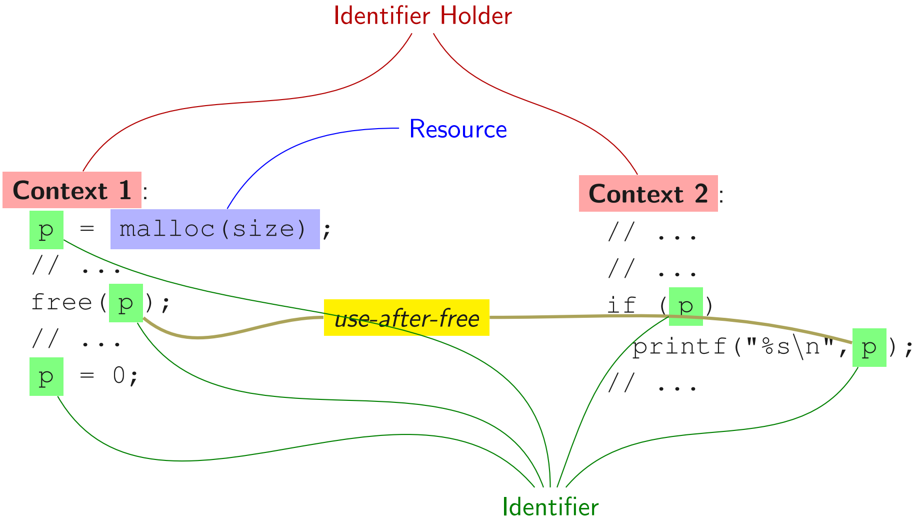
use-after-free

Context 2:

```
// ...
// ...
if (p)
  printf("%s\n", p);
// ...
```

Identifier

Use-after-free



Identifier Holder

Resource

Context 1:

```
p = malloc(size);
// ...
free(p);
// ...
p = 0;
```

use-after-free

Context 2:

```
// ...
// ...
if (p)
  printf("%s\n", p);
// ...
```

Identifier

Minimal Abstraction

A system **can be** vulnerable to Use-after-free **iff** the system has the concept of:

Minimal Abstraction

A system **can be** vulnerable to Use-after-free **iff** the system has the concept of:

1. resources,
2. resource identifiers,
3. and identifier holders.

Use-After-Free Vulnerable

A system **is** vulnerable to Use-after-free **iff**
the system allows to **silently exchange** resources.

Practical Example: Use-after-free

Use-after-free

```c
#include <stdio.h>

int *get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}

void secret() {
    int pins[] = {1337, 1589, 1346, 1470, 8846, 3478, 3669};
}

int main() {
    int *c = get_numbers();
    printf("%d %d %d %d %d %d %d\n", c[0], c[1], c[2], c[3], c[4], c[5], c[6]);
    secret();
    printf("%d %d %d %d %d %d %d\n", c[0], c[1], c[2], c[3], c[4], c[5], c[6]);
    return 0;
}
```

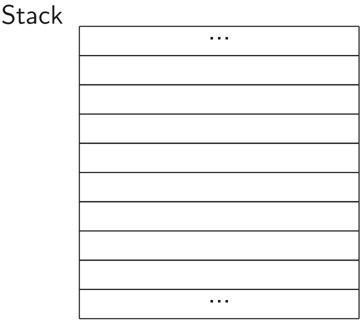
Use-after-free

```
% ./uaf-scope
1 2 4 8 16 32 64
1337 1589 1346 1470 8846 3478 3669
```

Practical Example Analysis: Use-after-free

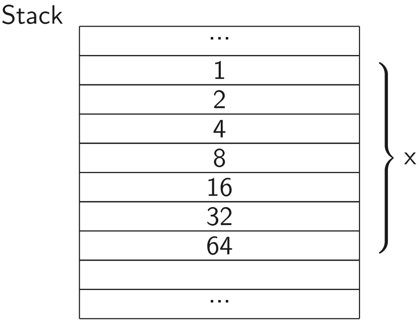
Use-after-free

```
int *get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}
void secret() {
    int pins[] = {1337, 1589, 1346,
        1470, 8846, 3478, 3669};
}
int main() {
    int *c = get_numbers();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
    secret();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
}
```



Use-after-free

```
int *get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}
void secret() {
    int pins[] = {1337, 1589, 1346,
        1470, 8846, 3478, 3669};
}
int main() {
    int *c = get_numbers();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
    secret();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
}
```


Stack

...

1

2

4

8

16

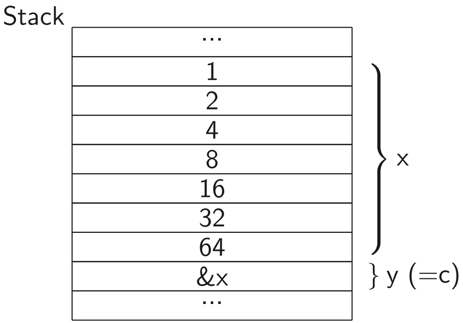
32

64

...

Use-after-free

```
int *get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}
void secret() {
    int pins[] = {1337, 1589, 1346,
        1470, 8846, 3478, 3669};
}
int main() {
    int *c = get_numbers();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
    secret();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
}
```



Stack

...

1

2

4

8

16

32

64

&x

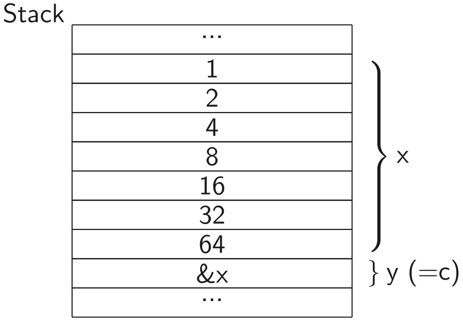
...

x

y (=c)

Use-after-free

```
int *get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}
void secret() {
    int pins[] = {1337, 1589, 1346,
        1470, 8846, 3478, 3669};
}
int main() {
    int *c = get_numbers();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
    secret();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
}
```



Stack

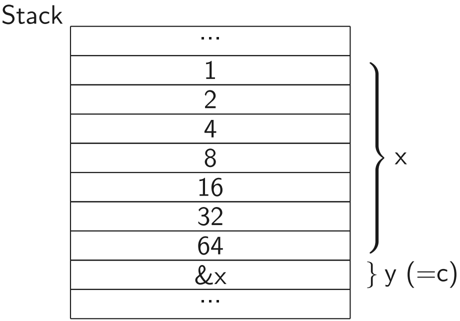
...
1
2
4
8
16
32
64
&x
...

} x

} y (=c)

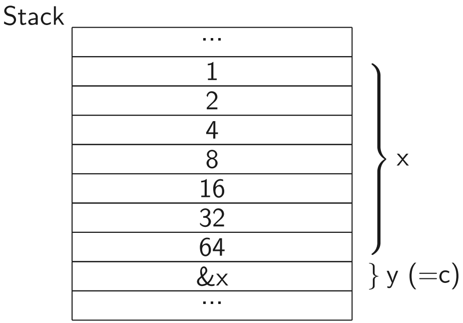
Use-after-free

```
int *get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}
void secret() {
    int pins[] = {1337, 1589, 1346,
        1470, 8846, 3478, 3669};
}
int main() {
    int *c = get_numbers();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
    secret();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
}
```



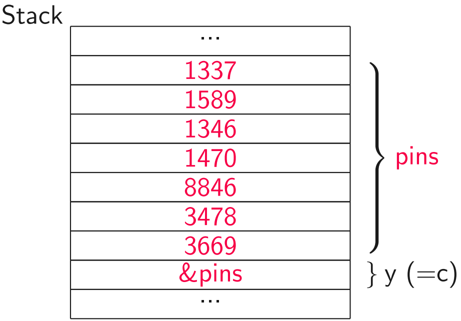
Use-after-free

```
int *get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}
void secret() {
    int pins[] = {1337, 1589, 1346,
        1470, 8846, 3478, 3669};
}
int main() {
    int *c = get_numbers();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
    secret();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
}
```



Use-after-free

```
int *get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}
void secret() {
    int pins[] = {1337, 1589, 1346,
        1470, 8846, 3478, 3669};
}
int main() {
    int *c = get_numbers();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
    secret();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
}
```



Stack

...

1337

1589

1346

1470

8846

3478

3669

&pins

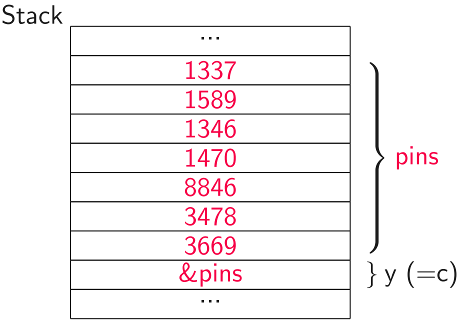
...

pins

} y (=c)

Use-after-free

```c
int *get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}
void secret() {
    int pins[] = {1337, 1589, 1346,
        1470, 8846, 3478, 3669};
}
int main() {
    int *c = get_numbers();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
    secret();
    printf("%d %d %d %d %d %d %d\n", c
        [0], c[1], c[2], c[3], c[4], c
        [5], c[6]);
}
```



Stack

...
1337
1589
1346
1470
8846
3478
3669
&pins
...

Practical Example Impact: Use-after-free

Use-after-free

- Stack frames are automatically destroyed

Use-after-free

- Stack frames are automatically destroyed
- However, references can still point to the stack frame

Use-after-free

- Stack frames are automatically destroyed
- However, references can still point to the stack frame
- Not easy to spot

Use-after-free

- Stack frames are automatically destroyed
- However, references can still point to the stack frame
- Not easy to spot
- Sometimes causes compiler warning, but not in this case

Use-after-free

- Stack frames are automatically destroyed
- However, references can still point to the stack frame
- Not easy to spot
- Sometimes causes compiler warning, but not in this case
- Attacker has access to confidential data of new stack frame

Fun Example: Use-after-free with Threads

Use-after-free Threads

```c
pthread_t tid;
void* thread(void* arg) {  printf("%s\n", (char*)arg); }

void start_thread() {
    char argument[64];
    strcpy(argument, "I’m a thread\n");
    pthread_create(&tid, NULL, thread, (void*)&argument);
}
void do_something() {
    char msg[64];
    strcpy(msg, "I’m NOT a thread\n");
}
int main() {
    start_thread();
    do_something();
    pthread_join(tid, NULL);
    return 0;
}
```

Use-after-free Threads

```
% ./uaf-thread
����
```

Use-after-free Threads

```
pthread_t tid;
void* thread(void* arg) {  printf("%s\n", (char*)arg); }

void start_thread() {
    char argument[64];
    strcpy(argument, "I'm a thread\n");
    pthread_create(&tid, NULL, thread, (void*)&argument);
}
void do_something() {
    char msg[64];
    sleep(1);
    strcpy(msg, "I'm NOT a thread\n");
    sleep(1);
}
int main() {
    start_thread();
    do_something();
    pthread_join(tid, NULL);
    return 0;
}
```

Use-after-free Threads

```
% ./uaf-thread
I'm a thread
```

Use-after-free Threads

```c
pthread_t tid;
void* thread(void* arg) {  printf("%s\n", (char*)arg); }

void start_thread() {
    char argument[64];
    strcpy(argument, "I’m a thread\n");
    pthread_create(&tid, NULL, thread, (void*)&argument);
}
void do_something() {
    char msg[64];
    strcpy(msg, "I’m NOT a thread\n");
    sleep(1);
}
int main() {
    start_thread();
    do_something();
    pthread_join(tid, NULL);
    return 0;
}
```

Use-after-free Threads

```
% ./uaf-thread
I'm NOT a thread
```

Practical Example: Use-after-free

Use-after-free

```c
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
    operation* io = (operation*)malloc(sizeof(operation));
    io->print = puts;
    io->print("Hallo ");
    free(io);

    if(argc > 1) {
        char* buffer = (char*)malloc(8);
        strncpy(buffer, argv[1], 7);
        io->print(buffer);
        free(buffer);
    }
    return 0;
}
```

Use-after-free

```
% gdb --args ./hello
(gdb) r
Starting program: /home/hello
Hallo
[Inferior 1 (process 7378) exited normally]
```

Use-after-free

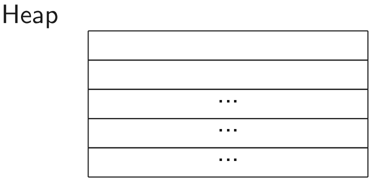
```
% gdb --args ./hello
(gdb) r
Starting program: /home/hello
Hallo
[Inferior 1 (process 7378) exited normally]
```

```
% gdb --args ./hello ABCD
(gdb) r
Starting program: /home/hello ABCD
Hallo

Program received signal SIGSEGV, Segmentation fault.
0x0000000044434241 in ?? ()
```

Practical Example Analysis: Use-after-free

Use-after-free

```
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
  operation* io = (operation*)malloc(
      sizeof(operation));
  io->print = puts;
  io->print("Hallo ");
  free(io);

  if(argc > 1) {
    char* buffer = (char*)malloc(8);
    strncpy(buffer, argv[1], 7);
    io->print(buffer);
    free(buffer);
  }
  return 0;
}
```


Heap

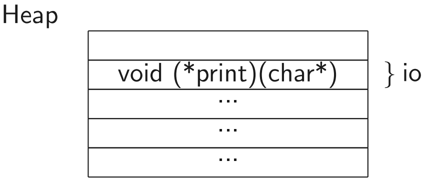
...

...

...

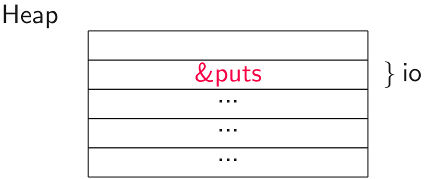
Use-after-free

```c
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
  operation* io = (operation*)malloc(
      sizeof(operation));
  io->print = puts;
  io->print("Hallo ");
  free(io);

  if(argc > 1) {
    char* buffer = (char*)malloc(8);
    strncpy(buffer, argv[1], 7);
    io->print(buffer);
    free(buffer);
  }
  return 0;
}
```



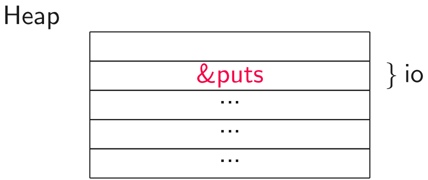
Use-after-free

```
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
  operation* io = (operation*)malloc(
      sizeof(operation));
  io->print = puts;
  io->print("Hallo ");
  free(io);

  if(argc > 1) {
    char* buffer = (char*)malloc(8);
    strncpy(buffer, argv[1], 7);
    io->print(buffer);
    free(buffer);
  }
  return 0;
}
```



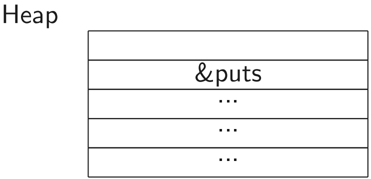
Use-after-free

```c
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
  operation* io = (operation*)malloc(
      sizeof(operation));
  io->print = puts;
  io->print("Hallo ");
  free(io);

  if(argc > 1) {
    char* buffer = (char*)malloc(8);
    strncpy(buffer, argv[1], 7);
    io->print(buffer);
    free(buffer);
  }
  return 0;
}
```



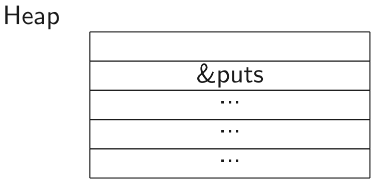
Use-after-free

```
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
  operation* io = (operation*)malloc(
      sizeof(operation));
  io->print = puts;
  io->print("Hallo ");
  free(io);

  if(argc > 1) {
    char* buffer = (char*)malloc(8);
    strncpy(buffer, argv[1], 7);
    io->print(buffer);
    free(buffer);
  }
  return 0;
}
```



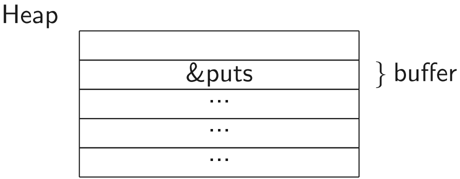
Use-after-free

```c
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
  operation* io = (operation*)malloc(
      sizeof(operation));
  io->print = puts;
  io->print("Hallo ");
  free(io);

  if(argc > 1) {
    char* buffer = (char*)malloc(8);
    strncpy(buffer, argv[1], 7);
    io->print(buffer);
    free(buffer);
  }
  return 0;
}
```



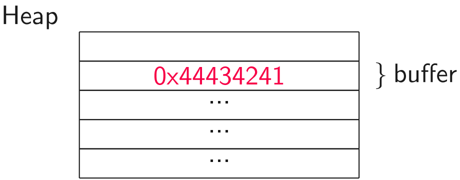
Use-after-free

```c
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
  operation* io = (operation*)malloc(
      sizeof(operation));
  io->print = puts;
  io->print("Hallo ");
  free(io);

  if(argc > 1) {
    char* buffer = (char*)malloc(8);
    strncpy(buffer, argv[1], 7);
    io->print(buffer);
    free(buffer);
  }
  return 0;
}
```



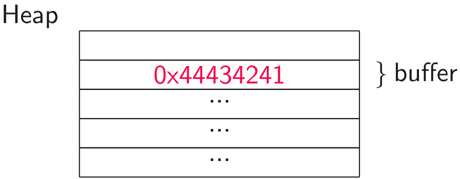
Use-after-free

```c
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
  operation* io = (operation*)malloc(
      sizeof(operation));
  io->print = puts;
  io->print("Hallo ");
  free(io);

  if(argc > 1) {
    char* buffer = (char*)malloc(8);
    strncpy(buffer, argv[1], 7);
    io->print(buffer);
    free(buffer);
  }
  return 0;
}
```



Use-after-free

```c
typedef struct {
    void (*print)(char*);
} operation;

int main(int argc, char* argv[]) {
  operation* io = (operation*)malloc(
      sizeof(operation));
  io->print = puts;
  io->print("Hallo ");
  free(io);

  if(argc > 1) {
    char* buffer = (char*)malloc(8);
    strncpy(buffer, argv[1], 7);
    io->print(buffer);
    free(buffer);
  }
  return 0;
}
```



Practical Example Impact: Use-after-free

Use-after-free

- Reference can point to different memory block or inside a memory block

Use-after-free

- Reference can point to different memory block or inside a memory block
- Using the reference corrupts valid memory

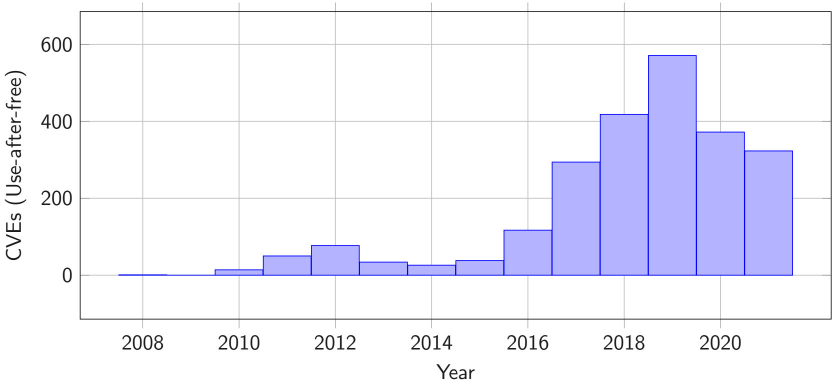
Use-after-free

- Reference can point to different memory block or inside a memory block
- Using the reference corrupts valid memory
- Allows to read possibly confidential data or overwrite data

Use-after-free

- Reference can point to different memory block or inside a memory block
- Using the reference corrupts valid memory
- Allows to read possibly confidential data or overwrite data
- Overwriting C++ object vtables allows to execute arbitrary code

Use-after-free - CVEs per year



Use-after-free Vulnerable Systems

Resource (R)	Resource Identifier (I_R)	Identifier Holder (H_I)
Memory buffer	Pointer / Address	Variables

Use-after-free Vulnerable Systems

Resource (R)	Resource Identifier (I_R)	Identifier Holder (H_I)
Memory buffer	Pointer / Address	Variables
Server	DNS entry / Domain	Links, databases, human memory

Use-after-free Vulnerable Systems

Resource (R)	Resource Identifier (I_R)	Identifier Holder (H_I)
Memory buffer	Pointer / Address	Variables
Server	DNS entry / Domain	Links, databases, human memory
Email account	Email address	Links, third-party websites, databases, address books, human memory

Use-after-free Vulnerable Systems

Resource (R)	Resource Identifier (I_R)	Identifier Holder (H_I)
Memory buffer	Pointer / Address	Variables
Server	DNS entry / Domain	Links, databases, human memory
Email account	Email address	Links, third-party websites, databases, address books, human memory
Twitter account	Twitter handle	Links, third-party websites, databases, human memory

Use-after-free Vulnerable Systems

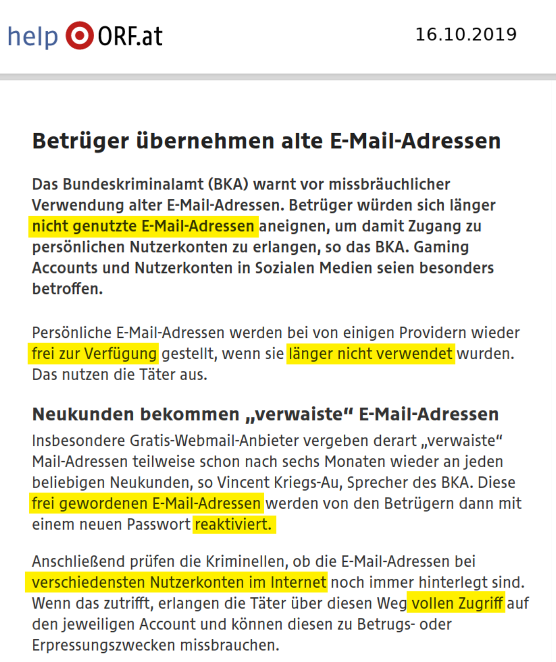
Resource (R)	Resource Identifier (I_R)	Identifier Holder (H_I)
Memory buffer	Pointer / Address	Variables
Server	DNS entry / Domain	Links, databases, human memory
Email account	Email address	Links, third-party websites, databases, address books, human memory
Twitter account	Twitter handle	Links, third-party websites, databases, human memory
Personal Phone	Phone Number	Personal and business address books, third-party websites, human memory

Use-after-free Vulnerable Systems

Resource (R)	Resource Identifier (I_R)	Identifier Holder (H_I)
Memory buffer	Pointer / Address	Variables
Server	DNS entry / Domain	Links, databases, human memory
Email account	Email address	Links, third-party websites, databases, address books, human memory
Twitter account	Twitter handle	Links, third-party websites, databases, human memory
Personal Phone	Phone Number	Personal and business address books, third-party websites, human memory
Mailbox	Address	Personal and business address books, human memory

Use-after-free Vulnerable Systems

Resource (R)	Resource Identifier (I_R)	Identifier Holder (H_I)
Memory buffer	Pointer / Address	Variables
Server	DNS entry / Domain	Links, databases, human memory
Email account	Email address	Links, third-party websites, databases, address books, human memory
Twitter account	Twitter handle	Links, third-party websites, databases, human memory
Personal Phone	Phone Number	Personal and business address books, third-party websites, human memory
Mailbox	Address	Personal and business address books, human memory
Employee	Office number	Human memory, business cards



help ORF.at 16.10.2019

Betrüger übernehmen alte E-Mail-Adressen

Das Bundeskriminalamt (BKA) warnt vor missbräuchlicher Verwendung alter E-Mail-Adressen. Betrüger würden sich länger nicht genutzte E-Mail-Adressen aneignen, um damit Zugang zu persönlichen Nutzerkonten zu erlangen, so das BKA. Gaming Accounts und Nutzerkonten in Sozialen Medien seien besonders betroffen.

Persönliche E-Mail-Adressen werden bei von einigen Providern wieder frei zur Verfügung gestellt, wenn sie länger nicht verwendet wurden. Das nutzen die Täter aus.

Neukunden bekommen „verwaiste“ E-Mail-Adressen

Insbesondere Gratis-Webmail-Anbieter vergeben derart „verwaiste“ Mail-Adressen teilweise schon nach sechs Monaten wieder an jeden beliebigen Neukunden, so Vincent Kriegs-Au, Sprecher des BKA. Diese frei gewordenen E-Mail-Adressen werden von den Betrügern dann mit einem neuen Passwort reaktiviert.

Anschließend prüfen die Kriminellen, ob die E-Mail-Adressen bei verschiedensten Nutzerkonten im Internet noch immer hinterlegt sind. Wenn das zutrifft, erlangen die Täter über diesen Weg vollen Zugriff auf den jeweiligen Account und können diesen zu Betrugs- oder Erpressungszwecken missbrauchen.

Double free

- Double free is similar to use-after-free

Double free

- Double free is similar to use-after-free
- Instead of referencing the memory after freeing, it is again freed

Double free

- Double free is similar to use-after-free
- Instead of referencing the memory after freeing, it is again freed
- Corrupts the internal memory management structures

Double free

- Double free is similar to use-after-free
- Instead of referencing the memory after freeing, it is again freed
- Corrupts the internal memory management structures
- Either crashes, corrupts memory, or returns same pointers for subsequent mallocs

Practical Example: Double free

Double free

```c
int main() {
    printf("Double free demo\n");
    char* b1 = malloc(16);
    char* b2 = malloc(16);
    char* b3 = malloc(16);
    free(b1);
    free(b2);
    free(b1);

    char* secret = malloc(16);
    strcpy(secret, "secret");
    char* dummy = malloc(16);

    char* buffer = malloc(16);
    printf("Should be empty (or garbage): \"%s\"\n", buffer);
    printf("&secret: %p, &buffer: %p\n", secret, buffer);
}
```

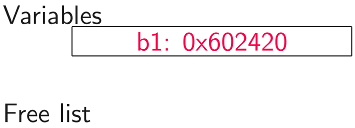
Double free

```
% ./doublefree
Double free demo
Should be empty (or garbage): "secret"
&secret: 0x2090420, &buffer: 0x2090420
```

Practical Example Analysis: Double free

Double free

```
int main() {
    printf("Double free demo\n");
    char* b1 = malloc(16);
    char* b2 = malloc(16);
    char* b3 = malloc(16);
    free(b1);
    free(b2);
    free(b1);

    char* secret = malloc(16);
    strcpy(secret, "secret");
    char* dummy = malloc(16);

    char* buffer = malloc(16);
    printf("Should be empty (or garbage
        ): \"%s\"\n", buffer);
    printf("&secret: %p, &buffer: %p\n"
        , secret, buffer);
}
```



Variables

b1: 0x602420

Free list

Double free

```c
int main() {
    printf("Double free demo\n");
    char* b1 = malloc(16);
    char* b2 = malloc(16);
    char* b3 = malloc(16);
    free(b1);
    free(b2);
    free(b1);

    char* secret = malloc(16);
    strcpy(secret, "secret");
    char* dummy = malloc(16);

    char* buffer = malloc(16);
    printf("Should be empty (or garbage
        ): \"%s\"\n", buffer);
    printf("&secret: %p, &buffer: %p\n"
        , secret, buffer);
}
```

Variables

b1: 0x602420
b2: 0x602440

Free list

Double free

```c
int main() {
    printf("Double free demo\n");
    char* b1 = malloc(16);
    char* b2 = malloc(16);
    char* b3 = malloc(16);
    free(b1);
    free(b2);
    free(b1);

    char* secret = malloc(16);
    strcpy(secret, "secret");
    char* dummy = malloc(16);

    char* buffer = malloc(16);
    printf("Should be empty (or garbage
        ): \"%s\"\n", buffer);
    printf("&secret: %p, &buffer: %p\n"
        , secret, buffer);
}
```

Variables

b1: 0x602420
b2: 0x602440
b3: 0x602460

Free list

Double free

```c
int main() {
    printf("Double free demo\n");
    char* b1 = malloc(16);
    char* b2 = malloc(16);
    char* b3 = malloc(16);
    free(b1);
    free(b2);
    free(b1);

    char* secret = malloc(16);
    strcpy(secret, "secret");
    char* dummy = malloc(16);

    char* buffer = malloc(16);
    printf("Should be empty (or garbage
        ): \"%s\"\n", buffer);
    printf("&secret: %p, &buffer: %p\n"
        , secret, buffer);
}
```

Variables

b1: 0x602420

b2: 0x602440

b3: 0x602460

Free list

0x602420 (b1)

Double free

```c
int main() {
    printf("Double free demo\n");
    char* b1 = malloc(16);
    char* b2 = malloc(16);
    char* b3 = malloc(16);
    free(b1);
    free(b2);
    free(b1);

    char* secret = malloc(16);
    strcpy(secret, "secret");
    char* dummy = malloc(16);

    char* buffer = malloc(16);
    printf("Should be empty (or garbage
        ): \"%s\"\n", buffer);
    printf("&secret: %p, &buffer: %p\n"
        , secret, buffer);
}
```

Variables

b1: 0x602420
b2: 0x602440
b3: 0x602460

Free list

0x602420 (b1)
0x602440 (b2)

Double free

```
int main() {
    printf("Double free demo\n");
    char* b1 = malloc(16);
    char* b2 = malloc(16);
    char* b3 = malloc(16);
    free(b1);
    free(b2);
    free(b1);

    char* secret = malloc(16);
    strcpy(secret, "secret");
    char* dummy = malloc(16);

    char* buffer = malloc(16);
    printf("Should be empty (or garbage
        ): \"%s\"\n", buffer);
    printf("&secret: %p, &buffer: %p\n"
        , secret, buffer);
}
```

Variables

b1: 0x602420
b2: 0x602440
b3: 0x602460

Free list

0x602420 (b1)
0x602440 (b2)
0x602420 (b1)

Double free

```c
int main() {
    printf("Double free demo\n");
    char* b1 = malloc(16);
    char* b2 = malloc(16);
    char* b3 = malloc(16);
    free(b1);
    free(b2);
    free(b1);

    char* secret = malloc(16);
    strcpy(secret, "secret");
    char* dummy = malloc(16);

    char* buffer = malloc(16);
    printf("Should be empty (or garbage
        ): \"%s\"\n", buffer);
    printf("&secret: %p, &buffer: %p\n"
        , secret, buffer);
}
```

Variables

b1: 0x602420
b2: 0x602440
b3: 0x602460
secret: 0x602420 (b1)

Free list

0x602440 (b2)
0x602420 (b1)

Double free

```c
int main() {
    printf("Double free demo\n");
    char* b1 = malloc(16);
    char* b2 = malloc(16);
    char* b3 = malloc(16);
    free(b1);
    free(b2);
    free(b1);

    char* secret = malloc(16);
    strcpy(secret, "secret");
    char* dummy = malloc(16);

    char* buffer = malloc(16);
    printf("Should be empty (or garbage
        ): \"%s\"\n", buffer);
    printf("&secret: %p, &buffer: %p\n"
        , secret, buffer);
}
```

Variables

b1: 0x602420
b2: 0x602440
b3: 0x602460
secret: 0x602420 (b1)

Free list

0x602440 (b2)
0x602420 (b1)

Double free

```c
int main() {
    printf("Double free demo\n");
    char* b1 = malloc(16);
    char* b2 = malloc(16);
    char* b3 = malloc(16);
    free(b1);
    free(b2);
    free(b1);

    char* secret = malloc(16);
    strcpy(secret, "secret");
    char* dummy = malloc(16);

    char* buffer = malloc(16);
    printf("Should be empty (or garbage
        ): \"%s\"\n", buffer);
    printf("&secret: %p, &buffer: %p\n"
        , secret, buffer);
}
```

Variables

b1: 0x602420
b2: 0x602440
b3: 0x602460
secret: 0x602420 (b1)
dummy: 0x602440 (b2)

Free list

0x602420 (b1)

Double free

```c
int main() {
    printf("Double free demo\n");
    char* b1 = malloc(16);
    char* b2 = malloc(16);
    char* b3 = malloc(16);
    free(b1);
    free(b2);
    free(b1);

    char* secret = malloc(16);
    strcpy(secret, "secret");
    char* dummy = malloc(16);

    char* buffer = malloc(16);
    printf("Should be empty (or garbage
        ): \"%s\"\n", buffer);
    printf("&secret: %p, &buffer: %p\n"
        , secret, buffer);
}
```

Variables

b1: 0x602420
b2: 0x602440
b3: 0x602460
secret: 0x602420 (b1)
dummy: 0x602440 (b2)
buffer: 0x602420 (b1)

Free list

Practical Example Impact: Double free

Double free

- Similar as use-after-free: two (different) references to the same memory location

Double free

- Similar as use-after-free: two (different) references to the same memory location
- Attacker can read confidential data

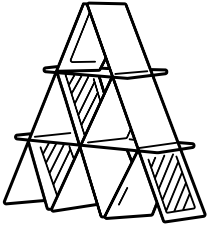
Double free

- Similar as use-after-free: two (different) references to the same memory location
- Attacker can read confidential data
- Memory can be corrupted

Double free

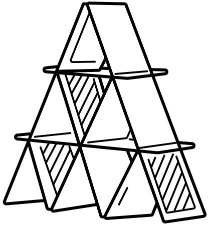
- Similar as use-after-free: two (different) references to the same memory location
- Attacker can read confidential data
- Memory can be corrupted
- If C++ object vtable in memory region, attacker gets arbitrary code execution

Exploitation Techniques



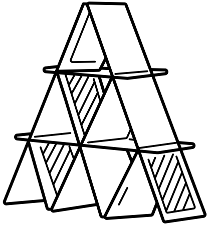
- Goals

Exploitation Techniques



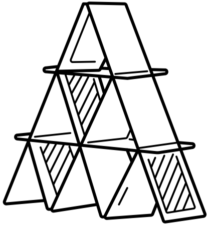
- Goals
  - Create overlapping chunks

Exploitation Techniques



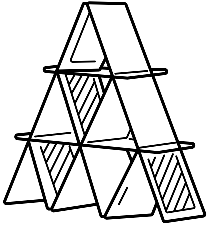
- Goals
  - Create overlapping chunks
  - Let malloc return arbitrary pointers

Exploitation Techniques



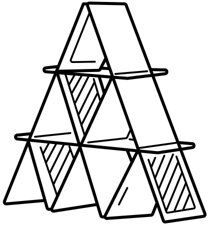
- Goals
  - Create overlapping chunks
  - Let malloc return arbitrary pointers
  - Use malloc to write to arbitrary addresses

Exploitation Techniques



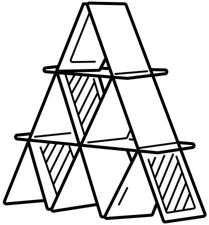
- Goals
  - Create overlapping chunks
  - Let malloc return arbitrary pointers
  - Use malloc to write to arbitrary addresses
- Common Techniques

Exploitation Techniques



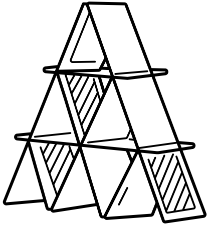
- Goals
  - Create overlapping chunks
  - Let malloc return arbitrary pointers
  - Use malloc to write to arbitrary addresses
- Common Techniques
  - Append fake chunks to free list

Exploitation Techniques



- Goals
  - Create overlapping chunks
  - Let malloc return arbitrary pointers
  - Use malloc to write to arbitrary addresses
- Common Techniques
  - Append fake chunks to free list
  - Overwrite metadata in free'd chunk (size/next pointer)

Exploitation Techniques



- Goals
  - Create overlapping chunks
  - Let malloc return arbitrary pointers
  - Use malloc to write to arbitrary addresses
- Common Techniques
  - Append fake chunks to free list
  - Overwrite metadata in free'd chunk (size/next pointer)
  - many many more (see Further Reading)

Format Strings

Format Strings

- C uses format strings to construct strings containing variables

Format Strings

- C uses format strings to construct strings containing variables
- Well known from `printf` or `fprintf`

```
printf("%d (dec) = 0x%x (hex)\n", 18, 18);
```

Format Strings

- C uses format strings to construct strings containing variables
- Well known from `printf` or `fprintf`

```
printf("%d (dec) = 0x%x (hex)\n", 18, 18);
```

- Format string parameters (`%d`, `%s`, ...) convert function parameters to strings

Format Strings

- C uses format strings to construct strings containing variables
- Well known from `printf` or `fprintf`

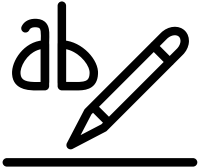
```
printf("%d (dec) = 0x%x (hex)\n", 18, 18);
```

- Format string parameters (`%d`, `%s`, ...) convert function parameters to strings
- Parameters are fetched from registers, and then from the stack (⇒ calling convention)

Format Strings

- What if the number of function parameters and format string parameters mismatch?

Format Strings



- What if the number of function parameters and format string parameters mismatch?
- `printf` trusts the format string (and the developer)

Format Strings

- What if the number of function parameters and format string parameters mismatch?
- `printf` trusts the format string (and the developer)
- `printf` is a variadic function, compiler does not care how many parameters

Format Strings

- What if the number of function parameters and format string parameters mismatch?
- `printf` trusts the format string (and the developer)
- `printf` is a variadic function, compiler does not care how many parameters
- If format string is constant, compiler could check it by understanding format strings
  - See `__attribute__((format(printf, ...)))`

Format Strings

- What if the number of function parameters and format string parameters mismatch?
- `printf` trusts the format string (and the developer)
- `printf` is a variadic function, compiler does not care how many parameters
- If format string is constant, compiler could check it by understanding format strings
  - See `__attribute__((format(printf, ...)))`
- In reality: no checks are performed (gcc only issues a warning)

Format Strings

- Usually no mismatch if developer writes the format string...

Format Strings

- Usually no mismatch if developer writes the format string...
- ...but if the attacker controls it:

```
printf(user_input);
```

Format Strings

- Usually no mismatch if developer writes the format string...
- ...but if the attacker controls it:

```
printf(user_input);
```

- If the user enters format string parameters, `printf` parses them although there are no function parameters

Practical Example: Format String

Format Strings

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    int secret_key = 0xdeadbeef;
    if(argc > 1)
        printf(argv[1]);
    return 0;
}
```

Format Strings

```
% ./echo "Test"
Test
% ./echo "Hello World"
Hello World
```

Format Strings

```
% ./echo "Test"
Test
% ./echo "Hello World"
Hello World
```

```
% ./echo "%p %p %p %p %p %p %p %p %p"
0x1 0x7fc3a4008780 0x7ffffff5 (nil) 0xb 0x7ffcb1b66db8
0x200400430 0x7ffcb1b66db0  0xdeadbeef00000000
```

Practical Example Analysis: Format String

Format Strings

```
printf("%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p ");
```

RSI (0x1)

Format Strings

```
printf("%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p ");
```

RSI (0x1)

RDX (0x7fc3a4008780)

Format Strings

```
printf("%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p ");
```

RSI (0x1)

RDX (0x7fc3a4008780)

RCX (0x7ffffff5)

Format Strings

```
printf("%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p ");
```

RSI (0x1)

RDX (0x7fc3a4008780)

RCX (0x7ffffff5)

R8 (nil)

Format Strings

```
printf("%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p ");
```

RSI	(0x1)
RDX	(0x7fc3a4008780)
RCX	(0x7ffffff5)
R8	(nil)
R9	(0xb)

Format Strings

```
printf("%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p ");
```

RSI	(0x1)
RDX	(0x7fc3a4008780)
RCX	(0x7ffffff5)
R8	(nil)
R9	(0xb)
[RSP]	(0x7ffcb1b66db8)

Format Strings

```
printf("%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p ");
```

RSI	(0x1)
RDX	(0x7fc3a4008780)
RCX	(0x7ffffff5)
R8	(nil)
R9	(0xb)
[RSP]	(0x7ffcb1b66db8)
[RSP + 0x8]	(0x200400430)

Format Strings

```
printf("%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p ");
```

RSI	(0x1)
RDX	(0x7fc3a4008780)
RCX	(0x7ffffff5)
R8	(nil)
R9	(0xb)
[RSP]	(0x7ffcb1b66db8)
[RSP + 0x8]	(0x200400430)
[RSP + 0x10]	(0x7ffcb1b66db0)

Format Strings

```
printf("%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p "
"%p ");
```

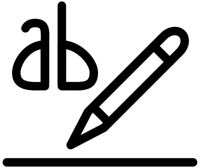
RSI	(0x1)
RDX	(0x7fc3a4008780)
RCX	(0x7ffffff5)
R8	(nil)
R9	(0xb)
[RSP]	(0x7ffcb1b66db8)
[RSP + 0x8]	(0x200400430)
[RSP + 0x10]	(0x7ffcb1b66db0)
[RSP + 0x18]	(0xdeadbeef00000000)

Practical Example Impact: Format String

Format Strings

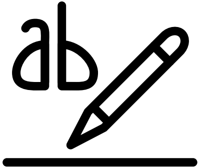
- A format string attack is possible if the user defines the format string

Format Strings



- A format string attack is possible if the user defines the format string
- It allows to easily read stack values

Format Strings



- A format string attack is possible if the user defines the format string
- It allows to easily read stack values
- Attacker might be able to read confidential data

Format Strings

- A format string attack is possible if the user defines the format string
- It allows to easily read stack values
- Attacker might be able to read confidential data
- Attacker can crash the program with enough `%s`

Format String - Short History

1989 First occured while fuzz testing, noted just as “interaction effect”

Format String - Short History

1989 First occured while fuzz testing, noted just as “interaction effect”

1999 First real format string bug in ProFTPD

Format String - Short History

1989 First occured while fuzz testing, noted just as “interaction effect”

1999 First real format string bug in ProFTPD

2000 First exploit (privilege escalation) published

Format String - Short History

1989 First occured while fuzz testing, noted just as “interaction effect”

1999 First real format string bug in ProFTPD

2000 First exploit (privilege escalation) published

2000 Exploits for many applications, including wu-ftpd (FTP), Qualcomm Popper (mail), Apache (webserver), OpenBSD, ...

Format Strings - Data Disclosure

- Format string parameters `%x` and similar (e.g., `%p`, `%d`, `%z`, ...) allow to read stack contents

Format Strings - Data Disclosure

- Format string parameters `%x` and similar (e.g., `%p`, `%d`, `%z`, ...) allow to read stack contents
- `%s` dereferences arbitrary addresses on the stack and outputs contents

Format Strings - Data Disclosure

- Format string parameters `%x` and similar (e.g., `%p`, `%d`, `%z`, ...) allow to read stack contents
- `%s` dereferences arbitrary addresses on the stack and outputs contents
- Encoding target address in format string allows to read arbitrary memory location

Format Strings - Data Disclosure

- Format string parameters `%x` and similar (e.g., `%p`, `%d`, `%z`, ...) allow to read stack contents
- `%s` dereferences arbitrary addresses on the stack and outputs contents
- Encoding target address in format string allows to read arbitrary memory location
- Thus, arbitrary addresses (both on stack and heap) can be disclosed

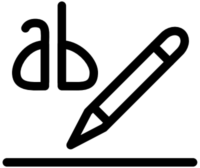
Format Strings - Data Disclosure

- Format string parameters `%x` and similar (e.g., `%p`, `%d`, `%z`, ...) allow to read stack contents
- `%s` dereferences arbitrary addresses on the stack and outputs contents
- Encoding target address in format string allows to read arbitrary memory location
- Thus, arbitrary addresses (both on stack and heap) can be disclosed
- What about manipulating data?

Format Strings - Data Manipulation

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

Format Strings - Data Manipulation



- 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.

Format Strings - Data Manipulation

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

man 3 printf

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

- Example:

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

Format Strings - Data Manipulation

- 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 argument. That argument shall be an int *, or variant whose size matches the (optionally) supplied integer length modifier.

- Example:

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

Prints `Wrote 12 characters`

Format Strings - Data Manipulation

- If there is an address on the stack, we can write to it

Format Strings - Data Manipulation

- If there is an address on the stack, we can write to it
- To write x to this address, just output x dummy bytes before using `%n`
- Example:

```c
int count;
printf("%1337s%n\n", "", &count);
printf("Wrote %d charachters\n", count);
```

Format Strings - Data Manipulation

- If there is an address on the stack, we can write to it
- To write x to this address, just output x dummy bytes before using `%n`
- Example:

```
int count;
printf("%1337s%n\n", "", &count);
printf("Wrote %d charachters\n", count);
```

Prints `Wrote 1337 characters`

Format Strings - Data Manipulation

- If there is an address on the stack, we can write to it
- To write x to this address, just output x dummy bytes before using `%n`
- Example:

```c
int count;
printf("%1337s%n\n", "", &count);
printf("Wrote %d charachters\n", count);
```

Prints `Wrote 1337 characters`

- The format string itself is also on the stack, so we can inject arbitrary addresses into the stack

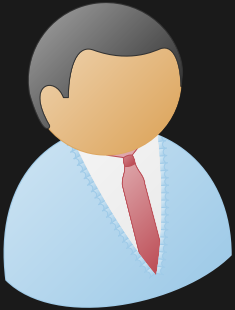
Fun Example: Format String Address Injection

Format String - Address Injection

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    char buffer[64];
    strcpy(buffer, argv[1]);
    printf(buffer);
}
```

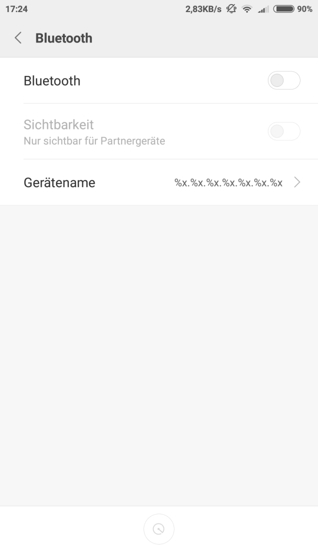
Format String - Address Injection

```
% valgrind ./format "ABCDABCD %p %p %p %p %p %p %p %n"
[...]
==17472== Invalid write of size 4
==17472==    at 0x4E89533: vfprintf (vfprintf.c:1631)
==17472==    by 0x4E8F898: printf (printf.c:33)
==17472==    by 0x40061E: main (printf.c:6)
==17472==  Address 0x4443424144434241 is not stack'd, malloc'd
           or (recently) free'd
==17472==
==17472==
==17472== Process terminating with default action of signal 11
          (SIGSEGV)
```



Real-world Example: Format String BMW 330i (CVE-2017-9212)

Format String BMW 330i (CVE-2017-9212)



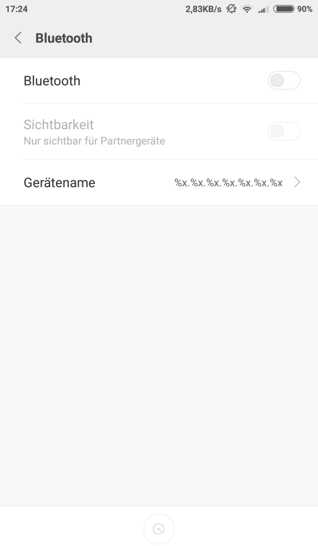
Bluetooth

Bluetooth

Sichtbarkeit
Nur sichtbar für Partnergeräte

Gerätename %x.%x.%x.%x.%x.%x.%x

Format String BMW 330i (CVE-2017-9212)



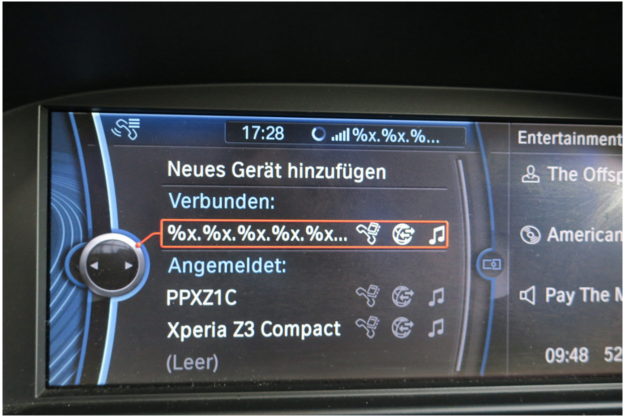
17:24 2,83KB/s 90%

Bluetooth

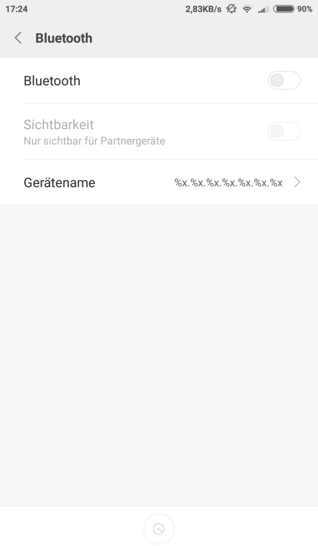
Bluetooth

Sichtbarkeit
Nur sichtbar für Partnergeräte

Gerätename %x.%x.%x.%x.%x.%x.%x



Format String BMW 330i (CVE-2017-9212)



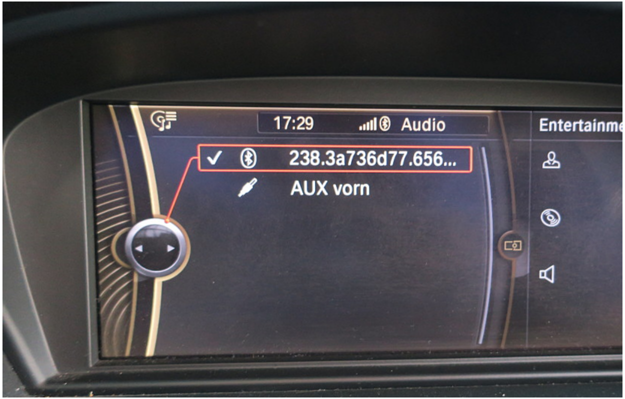
17:24 2,83KB/s 90%

Bluetooth

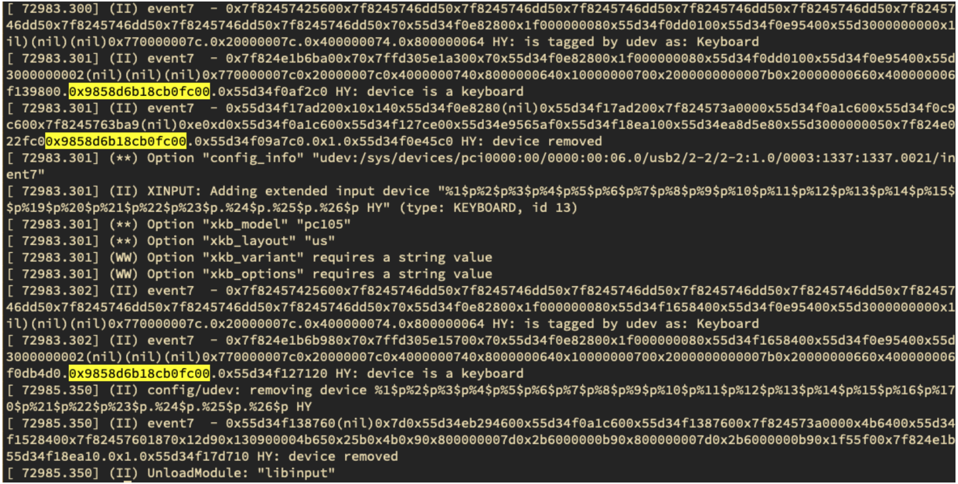
Bluetooth

Sichtbarkeit
Nur sichtbar für Partnergeräte

Gerätename %x.%x.%x.%x.%x.%x.%x



Format string vulnerability in evdev device handling (CVE-2022-1215)



```
[ 72983.300] (II) event7  - 0x7f82457425600x7f8245746dd50x7f8245746dd50x7f8245746dd50x7f8245746dd50x7f8245746dd50x7f82457
46dd50x7f8245746dd50x7f8245746dd50x7f8245746dd50x70x55d34f0e82800x1f000000080x55d34f0dd0100x55d34f0e95400x55d3000000000x1
il)(nil)(nil)0x770000007c.0x20000007c.0x400000074.0x800000064 HY: is tagged by udev as: Keyboard
[ 72983.301] (II) event7  - 0x7f824e1b6ba00x70x7ffd305e1a300x70x55d34f0e82800x1f000000080x55d34f0dd0100x55d34f0e95400x55d
3000000002(nil)(nil)(nil)0x770000007c0x20000007c0x4000000740x8000000640x10000000700x2000000000007b0x20000000660x400000006
f139800.0x9858d6b18cb0fc00.0x55d34f0af2c0 HY: device is a keyboard
[ 72983.301] (II) event7  - 0x55d34f17ad200x10x140x55d34f0e8280(nil)0x55d34f17ad200x7f824573a0000x55d34f0a1c600x55d34f0c9
c600x7f8245763ba9(nil)0xe0xd0x55d34f0a1c600x55d34f127ce00x55d34e9565af0x55d34f18ea100x55d34ea8d5e80x55d3000000050x7f824e0
22fc00x9858d6b18cb0fc00.0x55d34f09a7c0.0x1.0x55d34f0e45c0 HY: device removed
[ 72983.301] (**) Option "config_info" "udev:/sys/devices/pci0000:00/0000:00:06.0/usb2/2-2/2-2:1.0/0003:1337:1337.0021/in
ent7"
[ 72983.301] (II) XINPUT: Adding extended input device "%1$p%2$p%3$p%4$p%5$p%6$p%7$p%8$p%9$p%10$p%11$p%12$p%13$p%14$p%15$
$p%19$p%20$p%21$p%22$p%23$p.%24$p.%25$p.%26$p HY" (type: KEYBOARD, id 13)
[ 72983.301] (**) Option "xkb_model" "pc105"
[ 72983.301] (**) Option "xkb_layout" "us"
[ 72983.301] (WW) Option "xkb_variant" requires a string value
[ 72983.301] (WW) Option "xkb_options" requires a string value
[ 72983.302] (II) event7  - 0x7f82457425600x7f8245746dd50x7f8245746dd50x7f8245746dd50x7f8245746dd50x7f8245746dd50x7f82457
46dd50x7f8245746dd50x7f8245746dd50x7f8245746dd50x70x55d34f0e82800x1f000000080x55d34f1658400x55d34f0e95400x55d3000000000x1
il)(nil)(nil)0x770000007c.0x20000007c.0x400000074.0x800000064 HY: is tagged by udev as: Keyboard
[ 72983.302] (II) event7  - 0x7f824e1b6b980x70x7ffd305e15700x70x55d34f0e82800x1f000000080x55d34f1658400x55d34f0e95400x55d
3000000002(nil)(nil)(nil)0x770000007c0x20000007c0x4000000740x8000000640x10000000700x2000000000007b0x20000000660x400000006
f0db4d0.0x9858d6b18cb0fc00.0x55d34f127120 HY: device is a keyboard
[ 72985.350] (II) config/udev: removing device %1$p%2$p%3$p%4$p%5$p%6$p%7$p%8$p%9$p%10$p%11$p%12$p%13$p%14$p%15$p%16$p%17
0$p%21$p%22$p%23$p.%24$p.%25$p.%26$p HY
[ 72985.350] (II) event7  - 0x55d34f138760(nil)0x7d0x55d34eb294600x55d34f0a1c600x55d34f1387600x7f824573a0000x4b6400x55d34
f1528400x7f82457601870x12d90x130900004b650x25b0x4b0x90x800000007d0x2b6000000b90x800000007d0x2b6000000b90x1f55f00x7f824e1b
55d34f18ea10.0x1.0x55d34f17d710 HY: device removed
[ 72985.350] (II) UnloadModule: "libinput"
```

Format Strings Summary

- A format string attack is possible if the user can define the format string
- Not only in `printf`, but in the whole family (`fprintf`, `snprintf`, `vsprintf`, ...)
- It allows to read (or even manipulate)
  - arbitrary memory locations
  - itself (format strings are Turing complete)
- Easily preventable: never let the user control the format string

  -
    ```
    __attribute__((format(printf, 1, 2)))
    extern char *myFormatText2 (const char *, ...);
    ```
  - `gcc -Werror=format ...`

Challenge #3 - Format String

Find a format string to extract the binary's secret

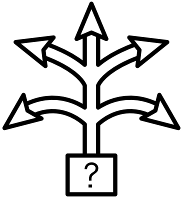
- The binary: `https://sasectf.student.iaik.tugraz.at/`
- Your format string has to extract the secret key (`<THE FLAG!>` in the sample code)
- Submitting the correct format string to the CTF system shows the real flag

Source

```c
char secret[15] __attribute__ ((section (".secret")));
int main(int argc, char* argv[]) {
    char buffer[16];
    printf("What do you want?\n");
    strcpy(secret, "<THE FLAG!>");
    fgets(buffer, 16, stdin);
    printf(buffer);
}
```

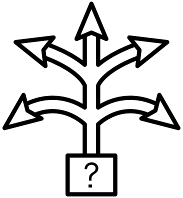
Type Confusion

Type Confusion



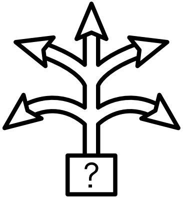
- A resource from one type is allocated, but later referenced as a different type

Type Confusion



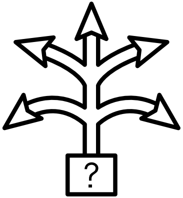
- A resource from one type is allocated, but later referenced as a different type
- No problem if the types are compatible ($\Rightarrow$ C++ polymorphism)

Type Confusion



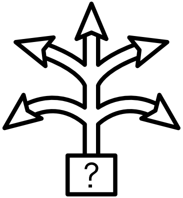
- A resource from one type is allocated, but later referenced as a different type
- No problem if the types are compatible (⇒ C++ polymorphism)
- C/C++ also allows casts to incompatible types, leading to logic errors

Type Confusion



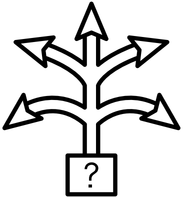
- A resource from one type is allocated, but later referenced as a different type
- No problem if the types are compatible (⇒ C++ polymorphism)
- C/C++ also allows casts to incompatible types, leading to logic errors
- Accesses can be out-of-bounds (⇒ buffer overflow), or leading to different control flow (⇒ vtables)

Type Confusion



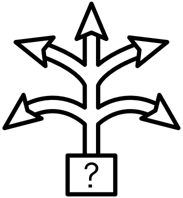
- C++ provides different types of casts

Type Confusion



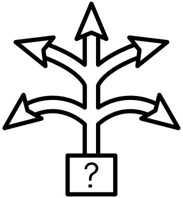
- C++ provides different types of casts
- dynamic_cast: Explicit type checks at runtime, but slow

Type Confusion



- C++ provides different types of casts
- `dynamic_cast`: Explicit type checks at runtime, but slow
- `static_cast`: Type check only at compile time, type confusion if runtime type is unexpected

Type Confusion



- C++ provides different types of casts
- `dynamic_cast`: Explicit type checks at runtime, but slow
- `static_cast`: Type check only at compile time, type confusion if runtime type is unexpected
- `reinterpret_cast`: Allows to explicitly break type checks

Practical Example: Type Confusion

Type Confusion

```cpp
#include <iostream>

class A {
  public: virtual const char* name() { return "A"; };
};
class B {
  public:  const char* name() { return "B"; };
  private: virtual const char* secret() { return "secret"; };
};

int main() {
  A* a = new A();
  std::cout << a->name() << std::endl;
  B* b = new B();
  std::cout << b->name() << std::endl;

  a = (A*)b;
  std::cout << a->name() << std::endl;
}
```

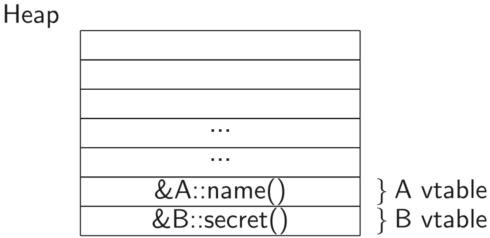
Type Confusion

```
% ./test
A
B
secret
```

Practical Example Analysis: Type Confusion

Type Confusion

```
class A {
  public: virtual const char* name()
    { return "A"; };
};
class B {
  public:  const char* name()
    { return "B"; };
  private: virtual const char* secret()
    { return "secret"; };
};
int main() {
  A* a = new A();
  std::cout << a->name() << std::endl;
  B* b = new B();
  std::cout << b->name() << std::endl;

  a = (A*)b;
  std::cout << a->name() << std::endl;
}
```



Heap

...

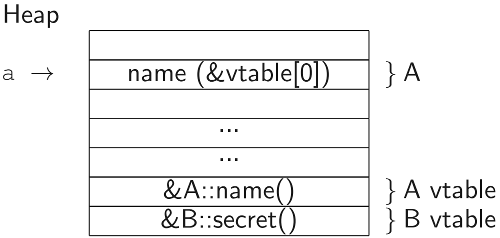
...

&A::name() } A vtable

&B::secret() } B vtable

Type Confusion

```
class A {
  public: virtual const char* name()
    { return "A"; };
};
class B {
  public:  const char* name()
    { return "B"; };
  private: virtual const char* secret()
    { return "secret"; };
};
int main() {
  A* a = new A();
  std::cout << a->name() << std::endl;
  B* b = new B();
  std::cout << b->name() << std::endl;

  a = (A*)b;
  std::cout << a->name() << std::endl;
}
```



Heap

a →

name (&vtable[0]) } A

...

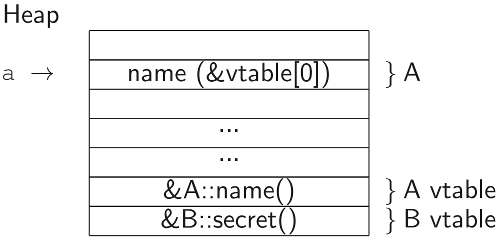
...

&A::name() } A vtable

&B::secret() } B vtable

Type Confusion

```
class A {
  public: virtual const char* name()
    { return "A"; };
};
class B {
  public:  const char* name()
    { return "B"; };
  private: virtual const char* secret()
    { return "secret"; };
};
int main() {
  A* a = new A();
  std::cout << a->name() << std::endl;
  B* b = new B();
  std::cout << b->name() << std::endl;

  a = (A*)b;
  std::cout << a->name() << std::endl;
}
```



Heap

a →

name (&vtable[0]) } A

...

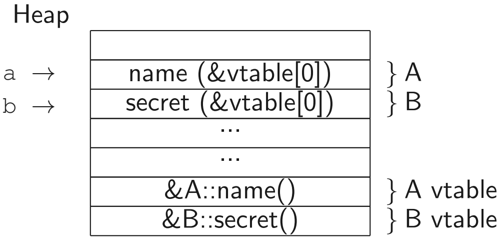
...

&A::name() } A vtable

&B::secret() } B vtable

Type Confusion

```
class A {
  public: virtual const char* name()
    { return "A"; };
};
class B {
  public:  const char* name()
    { return "B"; };
  private: virtual const char* secret()
    { return "secret"; };
};
int main() {
  A* a = new A();
  std::cout << a->name() << std::endl;
  B* b = new B();
  std::cout << b->name() << std::endl;

  a = (A*)b;
  std::cout << a->name() << std::endl;
}
```



Heap

a →

b →

name (&vtable[0]) } A

secret (&vtable[0]) } B

...

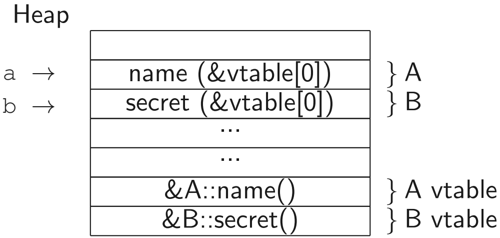
...

&A::name() } A vtable

&B::secret() } B vtable

Type Confusion

```
class A {
  public: virtual const char* name()
    { return "A"; };
};
class B {
  public:  const char* name()
    { return "B"; };
  private: virtual const char* secret()
    { return "secret"; };
};
int main() {
  A* a = new A();
  std::cout << a->name() << std::endl;
  B* b = new B();
  std::cout << b->name() << std::endl;

  a = (A*)b;
  std::cout << a->name() << std::endl;
}
```



Heap

a → name (&vtable[0]) } A

b → secret (&vtable[0]) } B

...

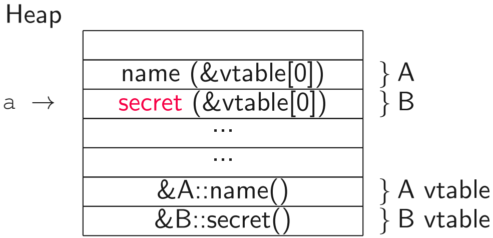
...

&A::name() } A vtable

&B::secret() } B vtable

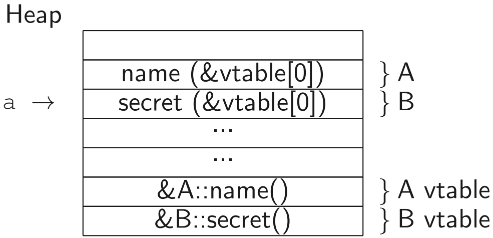
Type Confusion

```
class A {
  public: virtual const char* name()
    { return "A"; };
};
class B {
  public:  const char* name()
    { return "B"; };
  private: virtual const char* secret()
    { return "secret"; };
};
int main() {
  A* a = new A();
  std::cout << a->name() << std::endl;
  B* b = new B();
  std::cout << b->name() << std::endl;

  a = (A*)b;
  std::cout << a->name() << std::endl;
}
```



Type Confusion

```
class A {
  public: virtual const char* name()
    { return "A"; };
};
class B {
  public:  const char* name()
    { return "B"; };
  private: virtual const char* secret()
    { return "secret"; };
};
int main() {
  A* a = new A();
  std::cout << a->name() << std::endl;
  B* b = new B();
  std::cout << b->name() << std::endl;

  a = (A*)b;
  std::cout << a->name() << std::endl;
}
```



Heap

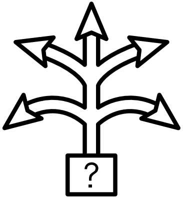
	name (&vtable[0])	} A
a →	secret (&vtable[0])	} B
	...	
	...	
	&A::name()	} A vtable
	&B::secret()	} B vtable

Type Confusion

```
% ./g++ vtables.h -fdump-lang-class && cat vtables.h.001l.class
Vtable for A                       Vtable for B
A::_ZTV1A: 3 entries               B::_ZTV1B: 3 entries
0      (int (*)(...))0             0      (int (*)(...))0
8      (int (*)(...))(& _ZTI1A)    8      (int (*)(...))(& _ZTI1B)
16     (int (*)(...))A::name       16     (int (*)(...))B::secret

Class A                            Class B
   size=8 align=8                  size=8 align=8
   base size=8 base align=8        base size=8 base align=8
A (0x0x7f4964ef0420) 0 nearly-empty B (0x0x7f4964ef04e0) 0 nearly-
    vptr=((& A::_ZTV1A) + 16)      vptr=((& B::_ZTV1B) + 16)
```

Practical Example Impact: Type Confusion

Type Confusion



- A type confusion happens if a pointer (or object) is cast to a wrong object
- It allows to
  - execute (arbitray) code
  - read/write out-of-bounds
  - crash the application
- Relatively new type of memory corruption

Type Confusion

- Type confusion bugs were exploited in many applications
  - Linux kernel (CVE-2022-34918)
  - Acrobat reader (CVE-2021-39841)
  - PHP (CVE-2016-3185)
  - Google Chrome (CVE-2022-2158, CVE-2022-1869,CVE-2022-1486,CVE-2022-1364,. . .)

Type Confusion

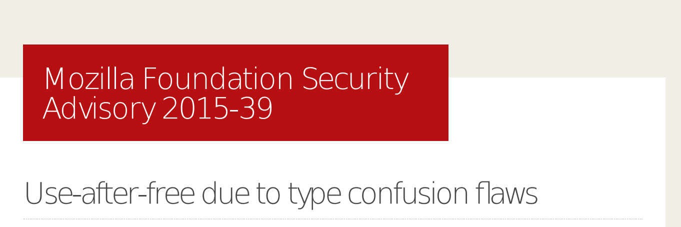
- Type confusion bugs were exploited in many applications
  - Linux kernel (CVE-2022-34918)
  - Acrobat reader (CVE-2021-39841)
  - PHP (CVE-2016-3185)
  - Google Chrome (CVE-2022-2158, CVE-2022-1869,CVE-2022-1486,CVE-2022-1364,. . .)
- Generally play an important role in browser exploits

Type Confusion

- Type confusion bugs were exploited in many applications
  - Linux kernel (CVE-2022-34918)
  - Acrobat reader (CVE-2021-39841)
  - PHP (CVE-2016-3185)
  - Google Chrome (CVE-2022-2158, CVE-2022-1869,CVE-2022-1486,CVE-2022-1364,. . .)
- Generally play an important role in browser exploits
- You can also be like Mozilla and combine them with other bugs:

Type Confusion

- Type confusion bugs were exploited in many applications
    - Linux kernel (CVE-2022-34918)
    - Acrobat reader (CVE-2021-39841)
    - PHP (CVE-2016-3185)
    - Google Chrome (CVE-2022-2158, CVE-2022-1869,CVE-2022-1486,CVE-2022-1364,. . .)
- Generally play an important role in browser exploits
- You can also be like Mozilla and combine them with other bugs:



Mozilla Foundation Security Advisory 2015-39

Use-after-free due to type confusion flaws

Invalid Memory Usage Summary

Invalid memory accesses...

Invalid Memory Usage Summary

Invalid memory accesses...

- are caused by different errors

Invalid Memory Usage Summary

Invalid memory accesses...

- are caused by different errors
- have varying impact

Invalid Memory Usage Summary

Invalid memory accesses...

- are caused by different errors
- have varying impact
- allow attacker to get full control over the system (more in the Exploit lecture)

Invalid Memory Usage Summary

Invalid memory accesses...

- are caused by different errors
- have varying impact
- allow attacker to get full control over the system (more in the Exploit lecture)
- are often harder to exploit than typical overflow bugs

Invalid Memory Usage Summary

Invalid memory accesses...

- are caused by different errors
- have varying impact
- allow attacker to get full control over the system (more in the Exploit lecture)
- are often harder to exploit than typical overflow bugs
- are not limited to C/C++

Memory Safety Violation Summary

Memory safety violations...

Memory Safety Violation Summary

Memory safety violations...

- are caused by a variety of errors

Memory Safety Violation Summary

Memory safety violations...

- are caused by a variety of errors
- are not limited to C/C++

Memory Safety Violation Summary

Memory safety violations...

- are caused by a variety of errors
- are not limited to C/C++
- are often hard to see in code

Memory Safety Violation Summary

Memory safety violations...

- are caused by a variety of errors
- are not limited to C/C++
- are often hard to see in code
- have very high impact

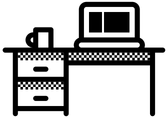
Memory Safety Violation Summary

Memory safety violations...

- are caused by a variety of errors
- are not limited to C/C++
- are often hard to see in code
- have very high impact
- are the base for exploits

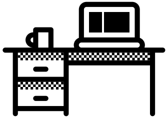
ENVIRONMENT

Environment



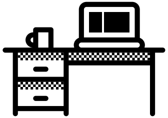
- Programs are always executed in some environment

Environment



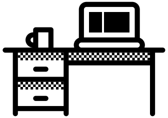
- Programs are always executed in some environment
- The environment is usually not fully known at compile time

Environment



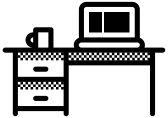
- Programs are always executed in some environment
- The environment is usually not fully known at compile time
- Defined by operating system, user, configurations, ...

Environment



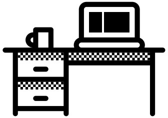
- Programs are always executed in some environment
- The environment is usually not fully known at compile time
- Defined by operating system, user, configurations, ...
- Environment can even change while the program is running

Environment



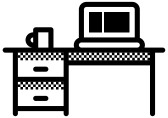
- Some bugs might not be exclusively in the binary

Environment



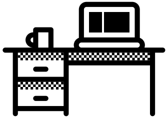
- Some bugs might not be exclusively in the binary
- They appear due to the program's interaction with the environment

Environment



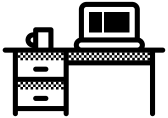
- Some bugs might not be exclusively in the binary
- They appear due to the program’s interaction with the environment
    - Environment variables

Environment



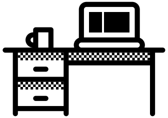
- Some bugs might not be exclusively in the binary
- They appear due to the program's interaction with the environment
  - Environment variables
  - Loader

Environment



- Some bugs might not be exclusively in the binary
- They appear due to the program's interaction with the environment
  - Environment variables
  - Loader
  - Access control

Environment



- Some bugs might not be exclusively in the binary
- They appear due to the program's interaction with the environment
  - Environment variables
  - Loader
  - Access control
- These factors have to be considered when writing programs

ENVIRONMENT
VARIABLES

Environment Variables

- Named values of the environment, usable by programs

Environment Variables

- Named values of the environment, usable by programs
- Each process has its own set (usually copy of the parent)

Environment Variables

- Named values of the environment, usable by programs
- Each process has its own set (usually copy of the parent)
- Provided by the envp pointer of `exec`

Environment Variables

- Named values of the environment, usable by programs
- Each process has its own set (usually copy of the parent)
- Provided by the envp pointer of `exec`
- Can also be set using `VARIABLE=VALUE` in most shells

Environment Variables

- Named values of the environment, usable by programs
- Each process has its own set (usually copy of the parent)
- Provided by the envp pointer of `exec`
- Can also be set using `VARIABLE=VALUE` in most shells
- Accessed using `setenv`/`getenv` in C/C++

Environment Variables

Some well-known environment variables

PATH Colon-separated list of folders to search for executables (e.g. `/usr/local/bin:/usr/bin:/bin:`)

HOME Path of user's home directory

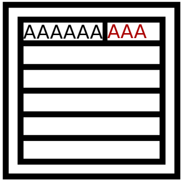
PWD The current directory

DISPLAY Identifier of the default X11 display (e.g. `:0`)

LANG Default locale (e.g. `en_US.UTF-8`)

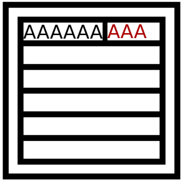
Environment Variables Problems

Buffer Overflow using Environment Variables



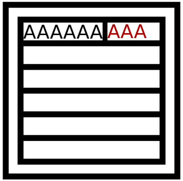
- Environment variables are strings ⇒ used with buffers

Buffer Overflow using Environment Variables



- Environment variables are strings ⇒ used with buffers
- Attacker controls length and content of environment variables

Buffer Overflow using Environment Variables



- Environment variables are strings ⇒ used with buffers
- Attacker controls length and content of environment variables
- Just a different form of user input

Practical Example: Buffer Overflow

Buffer Overflow (Environment Variable)

```c
#include <stdio.h>
#include <stdlib.h>

void greetings(int hello) {
    char buffer[32];
    if(hello) {
        sprintf(buffer, "Welcome %s", getenv("USER"));
    } else {
        sprintf(buffer, "Goodbye %s", getenv("USER"));
    }
    printf("%s\n", buffer);
}

int main() {
    greetings(1);
}
```

Buffer Overflow (Environment Variable)

```
% gdb ./env
(gdb) r
Starting program: /home/sasd/env
Welcome sasd
[Inferior 1 (process 14974) exited normally]
```

Buffer Overflow (Environment Variable)

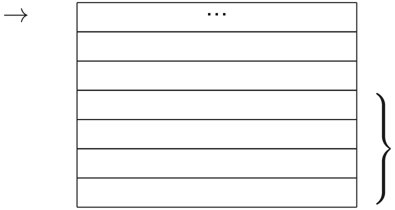
```
% gdb ./env
(gdb) r
Starting program: /home/sasd/env
Welcome sasd
[Inferior 1 (process 14974) exited normally]
```

```
% USER=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA; gdb ./env
(gdb) r
Starting program: /home/sasd/env
Welcome AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x000000000040061d in greetings (hello=1) at envovf.c:12
(gdb) bt
#0  0x000000000040061d in greetings (hello=1) at envovf.c:12
#1  0x4141414141414141 in ?? ()
```

Practical Example Analysis: Buffer Overflow

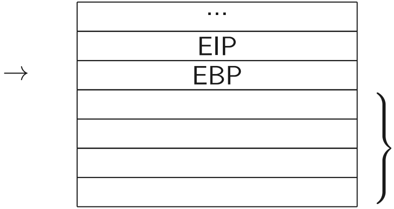
Buffer Overflow (Environment Variable)

```c
#include <stdio.h>
#include <stdlib.h>

void greetings(int hello) {
    char buffer[32];
    if(hello) {
        sprintf(buffer, "Welcome %s"
            , getenv("USER"));
    } else {
        sprintf(buffer, "Goodbye %s"
            , getenv("USER"));
    }
    printf("%s\n", buffer);
}

int main() {
    greetings(1);
}
```



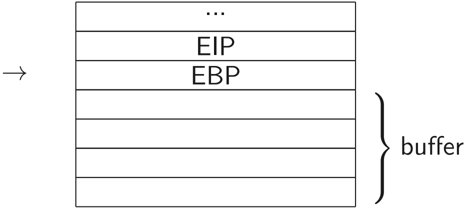
Buffer Overflow (Environment Variable)

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

void greetings(int hello) {
    char buffer[32];
    if(hello) {
        sprintf(buffer, "Welcome %s"
            , getenv("USER"));
    } else {
        sprintf(buffer, "Goodbye %s"
            , getenv("USER"));
    }
    printf("%s\n", buffer);
}

int main() {
    greetings(1);
}
```



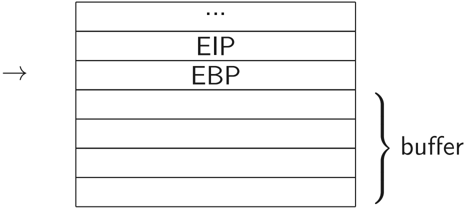
Buffer Overflow (Environment Variable)

```c
#include <stdio.h>
#include <stdlib.h>

void greetings(int hello) {
    char buffer[32];
    if(hello) {
        sprintf(buffer, "Welcome %s"
            , getenv("USER"));
    } else {
        sprintf(buffer, "Goodbye %s"
            , getenv("USER"));
    }
    printf("%s\n", buffer);
}

int main() {
    greetings(1);
}
```



Buffer Overflow (Environment Variable)

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

void greetings(int hello) {
    char buffer[32];
    if(hello) {
        sprintf(buffer, "Welcome %s"
            , getenv("USER"));
    } else {
        sprintf(buffer, "Goodbye %s"
            , getenv("USER"));
    }
    printf("%s\n", buffer);
}

int main() {
    greetings(1);
}
```



→

...

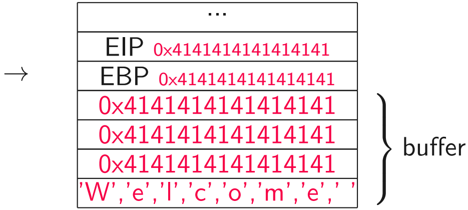
EIP

EBP

buffer

Buffer Overflow (Environment Variable)

```c
#include <stdio.h>
#include <stdlib.h>

void greetings(int hello) {
    char buffer[32];
    if(hello) {
        sprintf(buffer, "Welcome %s"
            , getenv("USER"));
    } else {
        sprintf(buffer, "Goodbye %s"
            , getenv("USER"));
    }
    printf("%s\n", buffer);
}

int main() {
    greetings(1);
}
```



→

...

EIP 0x4141414141414141

EBP 0x4141414141414141

0x4141414141414141

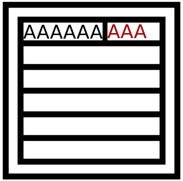
0x4141414141414141

0x4141414141414141

'W','e','l','c','o','m','e',' '

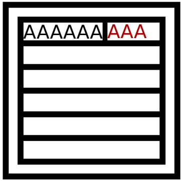
Practical Example Impact: Buffer Overflow

Buffer Overflow (Environment Variable)



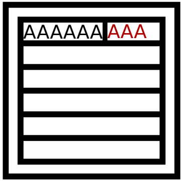
- Same impact as classical stack buffer overflow

Buffer Overflow (Environment Variable)



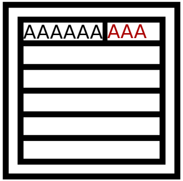
- Same impact as classical stack buffer overflow
- Attacker can jump to arbitrary location in memory

Buffer Overflow (Environment Variable)



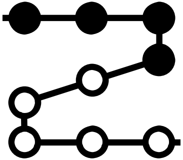
- Same impact as classical stack buffer overflow
- Attacker can jump to arbitrary location in memory
- Every function that is mapped in the address space can be executed

Buffer Overflow (Environment Variable)



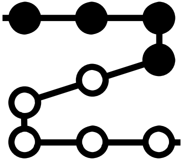
- Same impact as classical stack buffer overflow
- Attacker can jump to arbitrary location in memory
- Every function that is mapped in the address space can be executed
- Attacker has effectively full control over the program

PATH



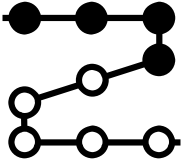
- If a binary to execute does not have full path (e.g. /bin/ls), folders in PATH variable are searched

PATH



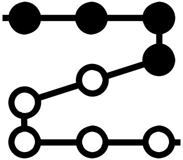
- If a binary to execute does not have full path (e.g. /bin/ls), folders in PATH variable are searched
- As soon as binary is found in one of these folders, it is executed

PATH



- If a binary to execute does not have full path (e.g. /bin/ls), folders in PATH variable are searched
- As soon as binary is found in one of these folders, it is executed
- Not only shell does that, but also `execlp`, `execvp`, and `system`

PATH



- If a binary to execute does not have full path (e.g. /bin/ls), folders in PATH variable are searched
- As soon as binary is found in one of these folders, it is executed
- Not only shell does that, but also `execlp`, `execvp`, and `system`
- Attacker might prepend folder to PATH variable

Fun Example: PATH manipulation

PATH manipulation

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    printf("Today: ");
    fflush(stdout);
    system("date");
}
```

PATH manipulation

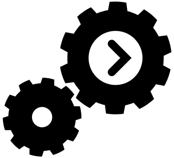
```
% cp /usr/games/fortune ./date
% ./today
Today: Fri Oct 27 13:17:34 CEST 2017
```

PATH manipulation

```
% cp /usr/games/fortune ./date
% ./today
Today: Fri Oct 27 13:17:34 CEST 2017
```

```
% export PATH=.:$PATH
% ./today
Today: It is so very hard to be an
on-your-own-take-care-of-yourself-because-there-is-no-one-else-
to-do-it-for-you grown-up.
```

LD_PRELOAD



- LD_PRELOAD is used by the dynamic linker/loader

LD_PRELOAD

- LD_PRELOAD is used by the dynamic linker/loader
- Contains one or more ELF shared object files

LD_PRELOAD

- LD_PRELOAD is used by the dynamic linker/loader
- Contains one or more ELF shared object files
- Object files are loaded before anything else

LD_PRELOAD

- LD_PRELOAD is used by the dynamic linker/loader
- Contains one or more ELF shared object files
- Object files are loaded before anything else
- Overwrites functions in other shared libraries

Fun Example: LD_PRELOAD

LD_PRELOAD

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char* argv[]) {
    char buffer[32];
    strcpy(buffer, "ultra secret password");
    if(getuid() == 0) {
        printf("Password: %s\n", buffer);
    } else {
        printf("Only root can get the password\n");
    }
}
```

LD_PRELOAD

```
% ./secret
Only root can get the password
```

LD_PRELOAD

```
% ./secret
Only root can get the password
```

```c
#include <sys/types.h>
uid_t getuid(void)
{
  return 0;
}
```

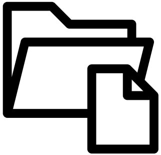
```
gcc -shared -fPIC getuid.c -o getuid.so
LD_PRELOAD=$PWD/getuid.so ./secret
Password: ultra secret password
```

Live Demo

Cheating in Tetris with LD_PRELOAD

FILE SYSTEM

File System



- File system does not only store the binaries

File System

- File system does not only store the binaries
- Keeps track of file permissions

File System

- File system does not only store the binaries
- Keeps track of file permissions
- Well-known permissions read, write, and execute for owner, group members, and others

File System

- File system does not only store the binaries
- Keeps track of file permissions
- Well-known permissions read, write, and execute for owner, group members, and others
- Lesser-known permissions setuid bit, setgid bit, and sticky bit

File System Pitfalls

setuid (SUID)

- setuid: short for “set user ID upon execution”

setuid (SUID)

- setuid: short for “set user ID upon execution”
- Runs the program with the rights of the owner (usually root) instead of the current user

setuid (SUID)

- setuid: short for “set user ID upon execution”
- Runs the program with the rights of the owner (usually root) instead of the current user
- Several standard tools have suid bit set (e.g. ping)

setuid (SUID)

- setuid: short for “set user ID upon execution”
- Runs the program with the rights of the owner (usually root) instead of the current user
- Several standard tools have suid bit set (e.g. ping)
- Exploiting a suid binary gives the attacker root privileges

setuid (SUID)

- setuid: short for “set user ID upon execution”
- Runs the program with the rights of the owner (usually root) instead of the current user
- Several standard tools have suid bit set (e.g. ping)
- Exploiting a suid binary gives the attacker root privileges
- `find / -perm -u=s -type f 2>/dev/null`

Linker as Interpreter

- Programs have the executable bit set

Linker as Interpreter

- Programs have the executable bit set
- Files without this bit cannot be executed

Linker as Interpreter

- Programs have the executable bit set
- Files without this bit cannot be executed
- Dynamic linker/loader is obviously executable

Linker as Interpreter

- Programs have the executable bit set
- Files without this bit cannot be executed
- Dynamic linker/loader is obviously executable
- It can be abused as interpreter

Fun Example: Linker as Interpreter

Linker as Interpreter

```
% ./hello
Hello World
```

Linker as Interpreter

```
% ./hello
Hello World
% chmod -x ./hello
% ./hello
bash: ./hello: Permission denied
```

Linker as Interpreter

```
% ./hello
Hello World
% chmod -x ./hello
% ./hello
bash: ./hello: Permission denied
```

```
% /lib64/ld-linux-x86-64.so.2 ./hello
Hello World
```

File TOCTTOU

- File system is asynchronous, files can change

File TOCTTOU

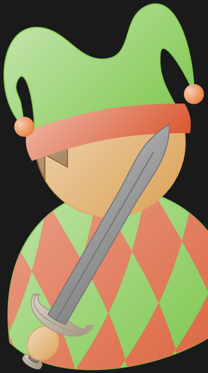
- File system is asynchronous, files can change
- If file can change between check and usage, this is a time-of-check-to-time-of-use (TOCTTOU) bug

File TOCTTOU

- File system is asynchronous, files can change
- If file can change between check and usage, this is a time-of-check-to-time-of-use (TOCTTOU) bug
- Problematic in combination with suid binaries

File TOCTTOU

- File system is asynchronous, files can change
- If file can change between check and usage, this is a time-of-check-to-time-of-use (TOCTTOU) bug
- Problematic in combination with suid binaries
- Program can be tricked to read different file by exchanging it



Fun Example: File TOCTTOU

File TOCTTOU

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
  char buffer[128];
  if(access(argv[1], R_OK) != 0) {
    printf("Access denied!\n");
    exit(0);
  }
  FILE* f = fopen(argv[1], "r");
  while(fgets(buffer, sizeof(buffer), f)) {
    printf("%s", buffer);
    memset(buffer, 0, sizeof(buffer));
  }
  fclose(f);
  return 0;
}
```

```
% ls -l supercat
-rwsrwsr-x 1 root root
  8776 Aug 27 21:58 supercat
% ./supercat /etc/shadow
Access denied!
% touch file
```

File TOCTTOU

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
  char buffer[128];
  if(access(argv[1], R_OK) != 0) {
    printf("Access denied!\n");
    exit(0);
  }
  FILE* f = fopen(argv[1], "r");
  while(fgets(buffer, sizeof(buffer), f)) {
    printf("%s", buffer);
    memset(buffer, 0, sizeof(buffer));
  }
  fclose(f);
  return 0;
}
```

```
% ls -l supercat
-rwsrwsr-x 1 root root
  8776 Aug 27 21:58 supercat
% ./supercat /etc/shadow
Access denied!
% touch file
% ./supercat file
```

File TOCTTOU

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
  char buffer[128];
  if(access(argv[1], R_OK) != 0) {
    printf("Access denied!\n");
    exit(0);
  }
  FILE* f = fopen(argv[1], "r");
  while(fgets(buffer, sizeof(buffer), f)) {
    printf("%s", buffer);
    memset(buffer, 0, sizeof(buffer));
  }
  fclose(f);
  return 0;
}
```

```
% ls -l supercat
-rwsrwsr-x 1 root root
  8776 Aug 27 21:58 supercat
% ./supercat /etc/shadow
Access denied!
% touch file
% ./supercat file
```

File TOCTTOU

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
  char buffer[128];
  if(access(argv[1], R_OK) != 0) {
    printf("Access denied!\n");
    exit(0);
  }
  FILE* f = fopen(argv[1], "r");
  while(fgets(buffer, sizeof(buffer), f)) {
    printf("%s", buffer);
    memset(buffer, 0, sizeof(buffer));
  }
  fclose(f);
  return 0;
}
```

```
% ls -l supercat
-rwsrwsr-x 1 root root
  8776 Aug 27 21:58 supercat
% ./supercat /etc/shadow
Access denied!
% touch file
% ./supercat file
% rm file
```

File TOCTTOU

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
  char buffer[128];
  if(access(argv[1], R_OK) != 0) {
    printf("Access denied!\n");
    exit(0);
  }
  FILE* f = fopen(argv[1], "r");
  while(fgets(buffer, sizeof(buffer), f)) {
    printf("%s", buffer);
    memset(buffer, 0, sizeof(buffer));
  }
  fclose(f);
  return 0;
}
```

```
% ls -l supercat
-rwsrwsr-x 1 root root
  8776 Aug 27 21:58 supercat
% ./supercat /etc/shadow
Access denied!
% touch file
% ./supercat file
% rm file
% ln -s /etc/shadow ./file
```

File TOCTTOU

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
  char buffer[128];
  if(access(argv[1], R_OK) != 0) {
    printf("Access denied!\n");
    exit(0);
  }
  FILE* f = fopen(argv[1], "r");
  while(fgets(buffer, sizeof(buffer), f)) {
    printf("%s", buffer);
    memset(buffer, 0, sizeof(buffer));
  }
  fclose(f);
  return 0;
}
```

```
% ls -l supercat
-rwsrwsr-x 1 root root
  8776 Aug 27 21:58 supercat
% ./supercat /etc/shadow
Access denied!
% touch file
% ./supercat file
% rm file
% ln -s /etc/shadow ./file
```

File TOCTTOU

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
  char buffer[128];
  if(access(argv[1], R_OK) != 0) {
    printf("Access denied!\n");
    exit(0);
  }
  FILE* f = fopen(argv[1], "r");
  while(fgets(buffer, sizeof(buffer), f)) {
    printf("%s", buffer);
    memset(buffer, 0, sizeof(buffer));
  }
  fclose(f);
  return 0;
}
```

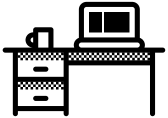
```
% ls -l supercat
-rwsrwsr-x 1 root root
  8776 Aug 27 21:58 supercat
% ./supercat /etc/shadow
Access denied!
% touch file
% ./supercat file
% rm file
% ln -s /etc/shadow ./file
```

File TOCTTOU

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]) {
  char buffer[128];
  if(access(argv[1], R_OK) != 0) {
    printf("Access denied!\n");
    exit(0);
  }
  FILE* f = fopen(argv[1], "r");
  while(fgets(buffer, sizeof(buffer), f)) {
    printf("%s", buffer);
    memset(buffer, 0, sizeof(buffer));
  }
  fclose(f);
  return 0;
}
```

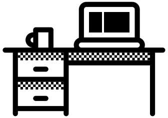
```
% ls -l supercat
-rwsrwsr-x 1 root root
  8776 Aug 27 21:58 supercat
% ./supercat /etc/shadow
Access denied!
% touch file
% ./supercat file
% rm file
% ln -s /etc/shadow ./file
root:!:17287:0:99999:7:::
```

Environment Summary



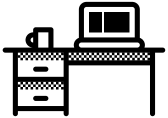
- User often controls the environment

Environment Summary



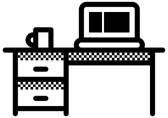
- User often controls the environment
- Never trust any input

Environment Summary



- User often controls the environment
- Never trust any input
- Consider environment properties as user input

Environment Summary



- User often controls the environment
- Never trust any input
- Consider environment properties as user input
- Environment can change during program execution → race conditions

Questions?

"It's the latest innovation in office safety.
When your computer crashes, an air bag is activated
so you won't bang your head in frustration."

Further Reading i

Jonathan Afek and Adi Sharabani.
Dangling pointer: Smashing the pointer for fun and profit.
Black Hat USA, 2007.

Azeria Labs.
Arm heap exploitation.
`https://azeria-labs.com/heap-exploitation`.

Daniel Gruss, Michael Schwarz, Matthias Wübbeling, Simon Guggi, Timo Malderle, Stefan More, and Moritz Lipp.
Use-after-freemail: Generalizing the use-after-free problem and applying it to email services.
In Proceedings of the 2018 on Asia Conference on Computer and Communications Security, 2018.

Further Reading ii

- Dhaval Kapil.
  Heap exploitation.
  https://heap-exploitation.dhavalkapil.com/attacks.
- Natalie Silvanovich, Dazed, (Type) Confused.
  One perfect bug: Exploiting type confusion in flash.
- scut / team teso.
  Exploiting format string vulnerabilities.
  https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf.
- Shellphish.
  how2heap.
  https://github.com/shellphish/how2heap.

Further Reading iii

- Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song.
  Sok: Eternal war in memory.
  In 2013 IEEE Symposium on Security and Privacy, 2013.

- Jinpeng Wei and Calton Pu.
  TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study.
  2005.