

Secure Software Development

Memory Corruption II & Environment

Daniel Gruss, Vedad Hadzic, Martin Schwarzl, Samuel Weiser

23.10.2020

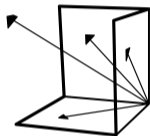
Winter 2020/21, www.iaik.tugraz.at

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

PREVIOUSLY ON

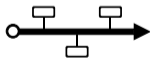
SSD

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



Overflow (last lecture)

- Stack overflow
- Heap overflow
- Integer overflow



Invalid Memory (this lecture)

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





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)



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)



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)



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)



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)



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)



Use-after-free



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



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



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



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

Context 1 :

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

Context 2 :

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

Context 1 :

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

Context 2 :

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

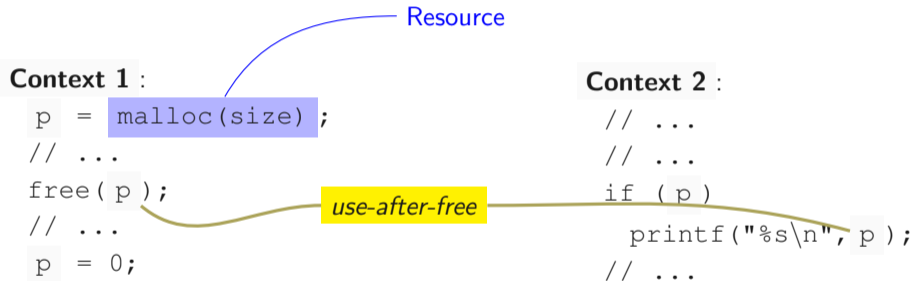
Context 1 :

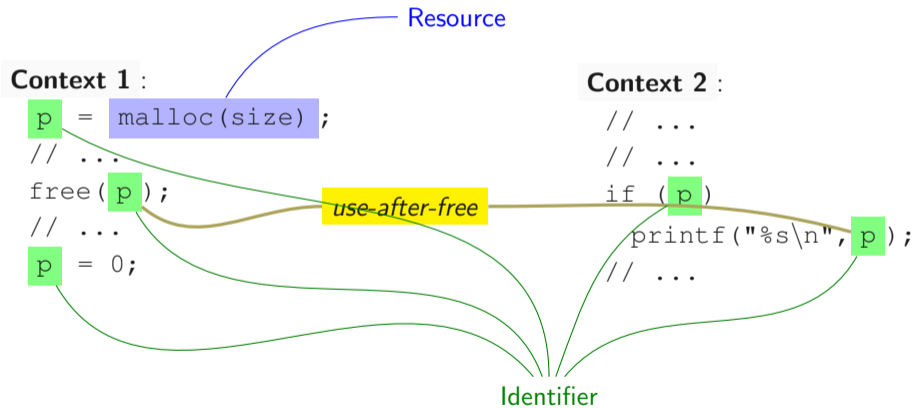
```
p = malloc(size) ;  
// ...  
free ( p ) ;  
// ...  
p = 0 ;
```

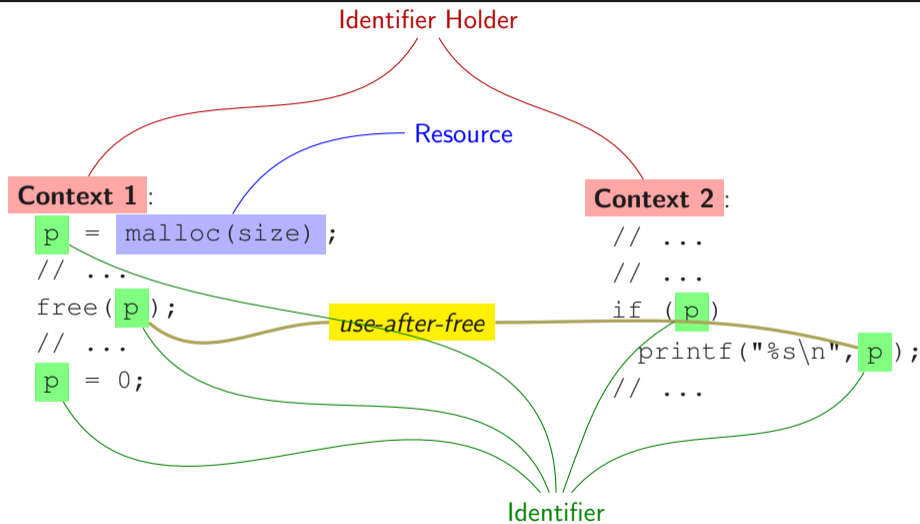
use-after-free

Context 2 :

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









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.



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.



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.



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.



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



Practical Example: Use-after-free



```
#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;
}
```



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




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

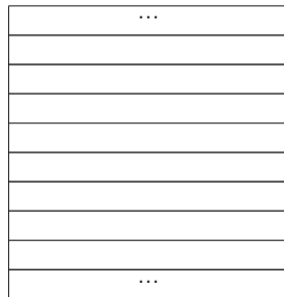


Practical Example Analysis: Use-after-free



```
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]);
}
```

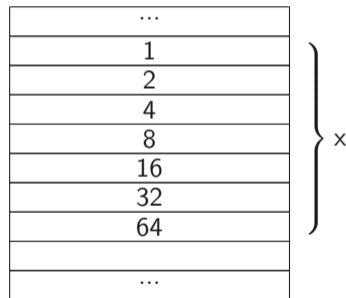
Stack





```
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



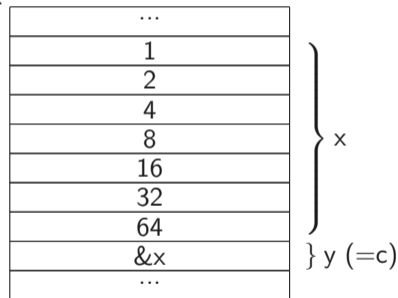


```
int* get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}

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



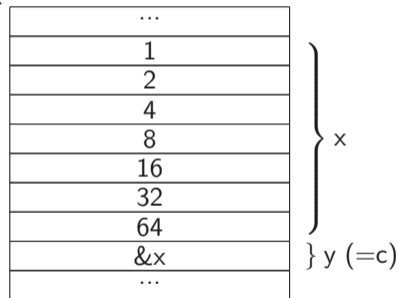


```
int* get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}

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]);
}
```

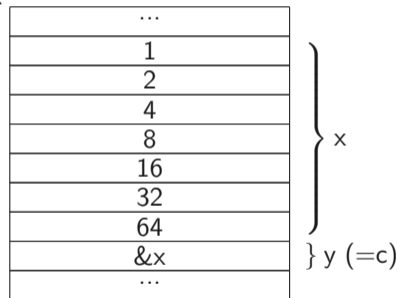
Stack





```
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]);
}
```

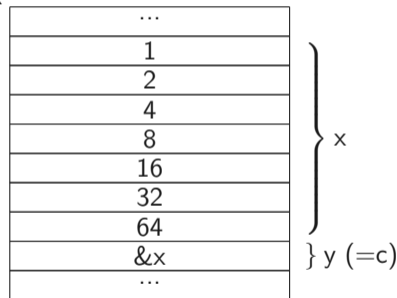
Stack





```
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]);
}
```

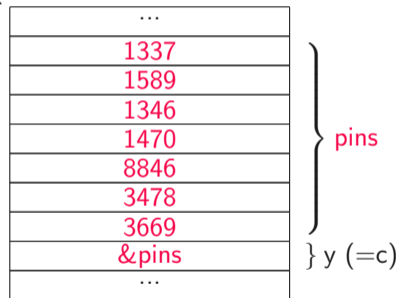
Stack





```
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



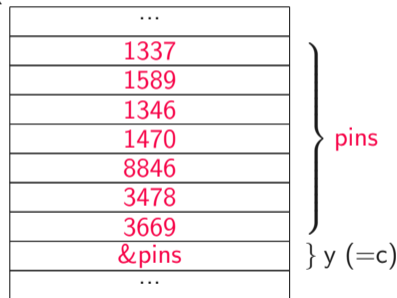


```
int* get_numbers() {
    int x[] = {1, 2, 4, 8, 16, 32, 64};
    int *y = x;
    return y;
}

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





Practical Example Impact: 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



- 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



- 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



- 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



- 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



```
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;
}
```



```
% ./uaf-thread
```





```
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;
}
```



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



```
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;
}
```



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



Practical Example: 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;
}
```



```
% 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 ?? ()
```



```
% 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

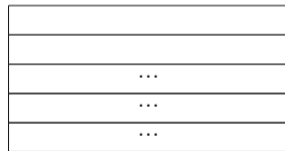


```
typedef struct {
    void (*print)(char*);
} operation;

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;
}
```

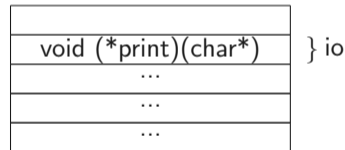
Heap





```
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



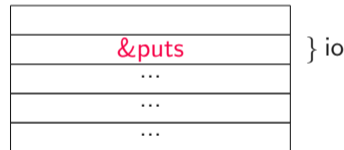


```
typedef struct {
    void (*print)(char*);
} operation;

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



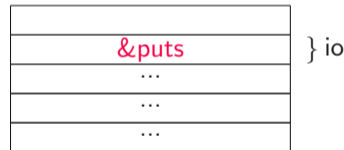


```
typedef struct {
    void (*print)(char*);
} operation;

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



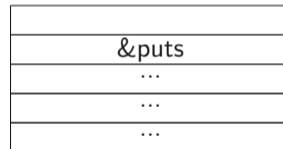


```
typedef struct {
    void (*print)(char*);
} operation;

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



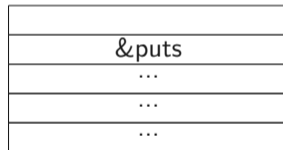


```
typedef struct {
    void (*print)(char*);
} operation;

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



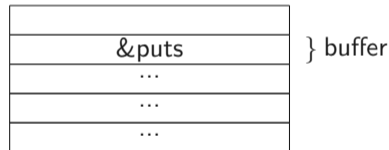


```
typedef struct {
    void (*print)(char*);
} operation;

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



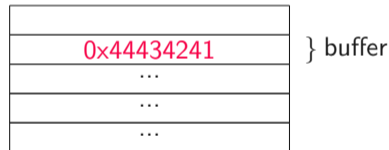


```
typedef struct {
    void (*print)(char*);
} operation;

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



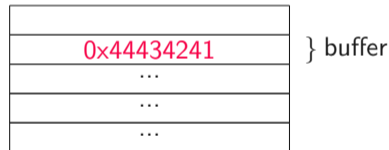


```
typedef struct {
    void (*print)(char*);
} operation;

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





Practical Example Impact: 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



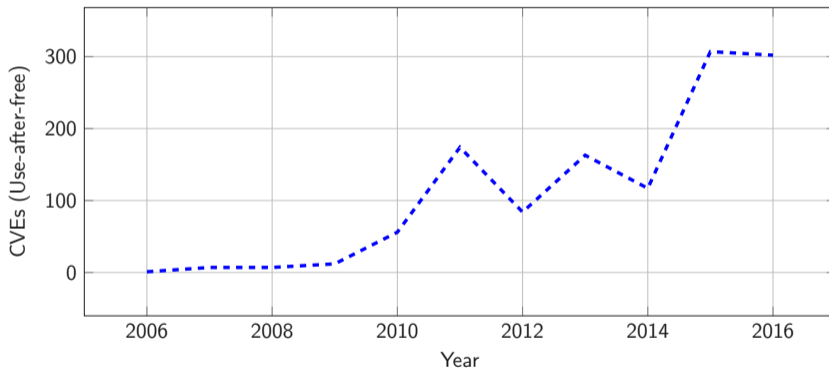
- Reference can point to **different memory** block or inside a memory block
- Using the reference **corrupts valid memory**
 - Allows to read possibly confidential data or overwrite data
 - Overwriting C++ object **vtables** allows to **execute arbitrary** code



- 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



- 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



Resource (R)

Resource Identifier (I_R)

Identifier Holder (H_I)

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

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

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

Resource (R)	Resource Identifier (I_R)	Identifier Holder (H_I)
Memory buffer	Pointer / Address	Variables
Server	DNS entry / (Sub-)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

Resource (R)	Resource Identifier (I_R)	Identifier Holder (H_I)
Memory buffer	Pointer / Address	Variables
Server	DNS entry / (Sub-)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

Resource (R)	Resource Identifier (I_R)	Identifier Holder (H_I)
Memory buffer	Pointer / Address	Variables
Server	DNS entry / (Sub-)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

Resource (R)	Resource Identifier (I_R)	Identifier Holder (H_I)
Memory buffer	Pointer / Address	Variables
Server	DNS entry / (Sub-)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

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** 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



- **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



- **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



- **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



```
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);
}
```



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



Practical Example Analysis: 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



```
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



```
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



```
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)



```
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)



```
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)



```
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)



```
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)



```
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)



```
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



- 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**



- 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**

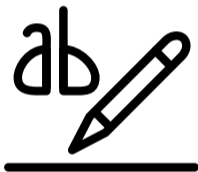


- Similar as use-after-free: two (different) references to the **same memory location**
- Attacker can read **confidential** data
- Memory can be **corrupted**
- If C++ object **vtable** in memory region, attacker gets **arbitrary code execution**



- 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**

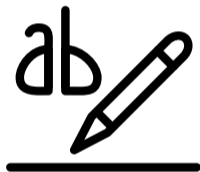
Format Strings



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

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

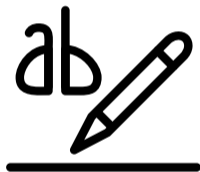
- Format string parameters (`%d`, `%s`, ...) convert function parameters to strings
- Parameters are fetched from **registers**, and then from the **stack** (\Rightarrow calling convention)



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

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

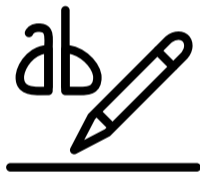
- Format string parameters (`%d`, `%s`, ...) convert function parameters to strings
- Parameters are fetched from **registers**, and then from the **stack** (\Rightarrow calling convention)



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

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

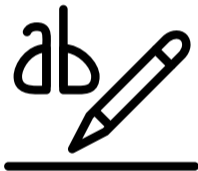
- Format string parameters (`%d`, `%s`, ...) convert function parameters to strings
- Parameters are fetched from **registers**, and then from the **stack** (\Rightarrow calling convention)



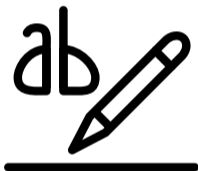
- C uses **format strings** to construct strings containing variables
- Well known from `printf` or `fprintf`

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

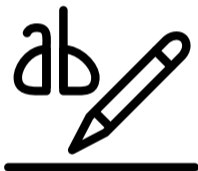
- Format string parameters (`%d`, `%s`, ...) convert function parameters to strings
- Parameters are fetched from **registers**, and then from the **stack** (\Rightarrow calling convention)



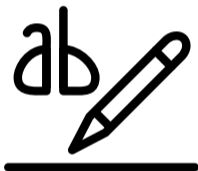
- What if the number of function parameters and format string parameters **mismatch**?
- `printf` **trusts** the format string (and the developer)
- `printf` is a **variadic function**, compiler does not care how many parameters
- If format string is **constant**, compiler **could check** it by understanding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



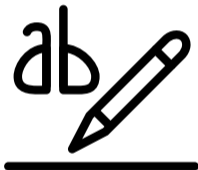
- What if the number of function parameters and format string parameters **mismatch**?
- `printf` **trusts** the format string (and the developer)
- `printf` is a **variadic function**, compiler does not care how many parameters
- If format string is **constant**, compiler **could check** it by understanding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



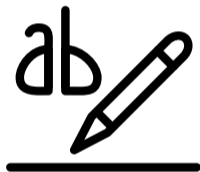
- What if the number of function parameters and format string parameters **mismatch**?
- `printf` **trusts** the format string (and the developer)
- `printf` is a **variadic function**, compiler does not care how many parameters
- If format string is **constant**, compiler **could check** it by understanding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



- 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
- In reality: **no checks** are performed (gcc only issues a **warning**)



- 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
- In reality: **no checks** are performed (gcc only issues a **warning**)



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

- ...but if the **attacker controls** it:

```
printf(user_input);
```

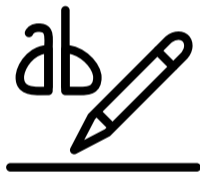
- If the user **enters format string parameters**, `printf` parses them although there are no function parameters



- 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



- 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



```
#include <stdio.h>

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



```
% ./echo "Test"
```

```
Test
```

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

```
Hello World
```

```
% ./echo "%p %p %p %p %p %p %p %p %p"
```

```
0x1 0x7fc3a4008780 0x7fffffff5 (nil) 0xb 0x7ffcb1b66db8
```

```
0x200400430 0x7ffcb1b66db0 0xdeadbeef0000000
```



```
% ./echo "Test"
```

```
Test
```

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

```
Hello World
```

```
% ./echo "%p %p %p %p %p %p %p %p %p %p"
```

```
0x1 0x7fc3a4008780 0x7fffffff5 (nil) 0xb 0x7ffcb1b66db8
```

```
0x200400430 0x7ffcb1b66db0 0xdeadbeef0000000
```



Practical Example Analysis: Format String



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

RSI (0x1)

RDX (0x7fc3a4008780)

RCX (0x7fffffff5)

R8 (nil)

R9 (0xb)

[RSP] (0x7ffcb1b66db8)

[RSP + 0x8] (0x200400430)

[RSP + 0x10] (0x7ffcb1b66db0)

[RSP + 0x18] (0xdeadbeef0000000)



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

RSI (0x1)

RDX (0x7fc3a4008780)

RCX (0x7fffffff5)

R8 (nil)

R9 (0xb)

[RSP] (0x7ffc1b66db8)

[RSP + 0x8] (0x200400430)

[RSP + 0x10] (0x7ffc1b66db0)

[RSP + 0x18] (0xdeadbeef0000000)



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

RSI (0x1)

RDX (0x7fc3a4008780)

RCX (0x7fffffff5)

R8 (nil)

R9 (0xb)

[RSP] (0x7ffcb1b66db8)

[RSP + 0x8] (0x200400430)

[RSP + 0x10] (0x7ffcb1b66db0)

[RSP + 0x18] (0xdeadbeef0000000)



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

RSI (0x1)

RDX (0x7fc3a4008780)

RCX (0x7fffffff5)

R8 (nil)

R9 (0xb)

[RSP] (0x7ffc1b66db8)

[RSP + 0x8] (0x200400430)

[RSP + 0x10] (0x7ffc1b66db0)

[RSP + 0x18] (0xdeadbeef0000000)



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

RSI (0x1)

RDX (0x7fc3a4008780)

RCX (0x7fffffff5)

R8 (nil)

R9 (0xb)

[RSP] (0x7ffcb1b66db8)

[RSP + 0x8] (0x200400430)

[RSP + 0x10] (0x7ffcb1b66db0)

[RSP + 0x18] (0xdeadbeef0000000)



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

RSI (0x1)

RDX (0x7fc3a4008780)

RCX (0x7fffffff5)

R8 (nil)

R9 (0xb)

[RSP] (0x7ffcb1b66db8)

[RSP + 0x8] (0x200400430)

[RSP + 0x10] (0x7ffcb1b66db0)

[RSP + 0x18] (0xdeadbeef0000000)



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

RSI (0x1)

RDX (0x7fc3a4008780)

RCX (0x7fffffff5)

R8 (nil)

R9 (0xb)

[RSP] (0x7ffcb1b66db8)

[RSP + 0x8] (0x200400430)

[RSP + 0x10] (0x7ffcb1b66db0)

[RSP + 0x18] (0xdeadbeef0000000)



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

RSI (0x1)

RDX (0x7fc3a4008780)

RCX (0x7fffffff5)

R8 (nil)

R9 (0xb)

[RSP] (0x7ffcb1b66db8)

[RSP + 0x8] (0x200400430)

[RSP + 0x10] (0x7ffcb1b66db0)

[RSP + 0x18] (0xdeadbeef0000000)



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

RSI (0x1)

RDX (0x7fc3a4008780)

RCX (0x7fffffff5)

R8 (nil)

R9 (0xb)

[RSP] (0x7ffcb1b66db8)

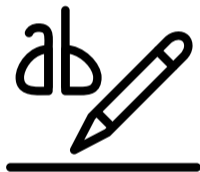
[RSP + 0x8] (0x200400430)

[RSP + 0x10] (0x7ffcb1b66db0)

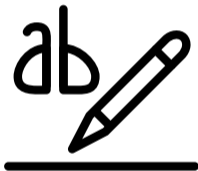
[RSP + 0x18] (0xdeadbeef00000000)



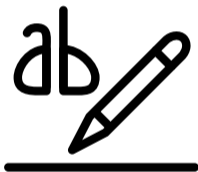
Practical Example Impact: Format String



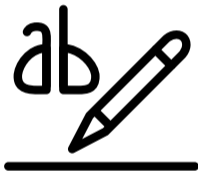
- A format string attack is possible if the **user** defines the **format string**
- It allows to easily read **stack values**
- Attacker might be able to read confidential data
- Attacker can crash the program with enough `%s`



- 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



- 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`



- 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`



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-ftp (FTP),
Qualcomm Popper (mail), Apache (webserver), OpenBSD, ...



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-ftp (FTP),
Qualcomm Popper (mail), Apache (webserver), OpenBSD, ...



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

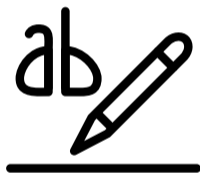
1999 First real format string **bug** in ProFTPD

2000 First exploit (privilege escalation) published

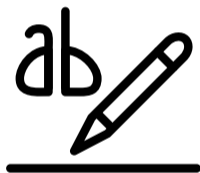
2000 Exploits for many applications, including wu-ftpd (FTP),
Qualcomm Popper (mail), Apache (webserver), OpenBSD, ...



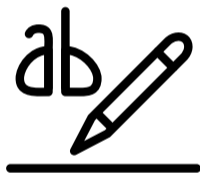
- 1989 First **occured** while fuzz testing, noted just as “interaction effect”
- 1999 First real format string **bug** in ProFTPD
- 2000 First exploit (privilege escalation) published
- 2000 Exploits for many applications, including wu-ftp (FTP), Qualcomm Popper (mail), Apache (webserver), OpenBSD, ...



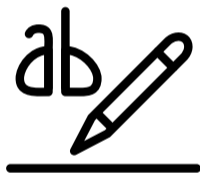
- Format string parameters `%x` and similar (e.g., `%p`, `%d`, `%z`, ...) allow to read **stack contents**
- `%s` **dereferences** arbitrary addresses on the stack and outputs contents
- Encoding target address in format string allows to read **arbitrary memory location**
- Thus, arbitrary addresses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



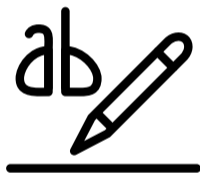
- Format string parameters `%x` and similar (e.g., `%p`, `%d`, `%z`, ...) allow to read **stack contents**
- `%s` **dereferences** arbitrary addresses on the stack and outputs contents
- Encoding target address in format string allows to read **arbitrary memory location**
- Thus, arbitrary addresses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



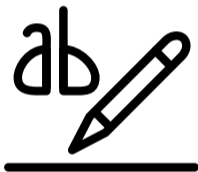
- Format string parameters `%x` and similar (e.g., `%p`, `%d`, `%z`, ...) allow to read **stack contents**
- `%s` **dereferences** arbitrary addresses on the stack and outputs contents
- Encoding target address in format string allows to read **arbitrary memory location**
- Thus, arbitrary addresses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



- Format 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 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?



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

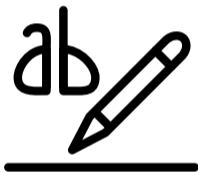
man 3 printf

`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



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

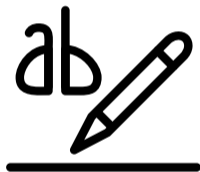
man 3 printf

`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



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

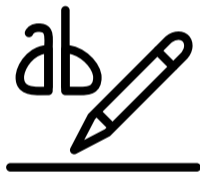
man 3 printf

`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



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

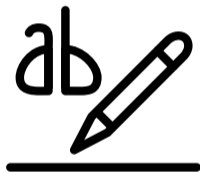
man 3 printf

`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

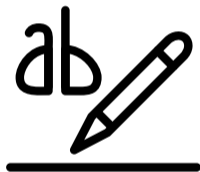


- If there is an **address** on the stack, we can **write** to it
- To write x to this address, just output x **dummy bytes** before using `%n`
- Example:

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

Prints `Wrote 1337 characters`

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

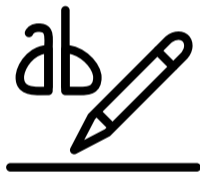


- If there is an **address** on the stack, we can **write** to it
- To write x to this address, just output x **dummy bytes** before using `%n`
- Example:

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

Prints wrote 1337 characters

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

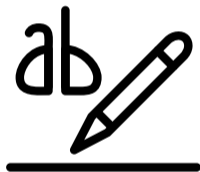


- If there is an **address** on the stack, we can **write** to it
- To write x to this address, just output x **dummy bytes** before using `%n`
- Example:

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

Prints Wrote 1337 characters

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



- 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 characters\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



```
#include <stdio.h>

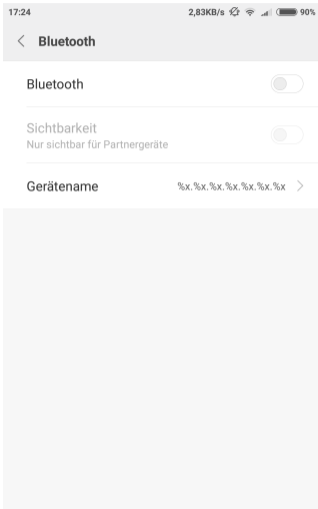
int main(int argc, char* argv[]) {
    char buffer[64];
    strcpy(buffer, argv[1]);
    printf(buffer);
}
```

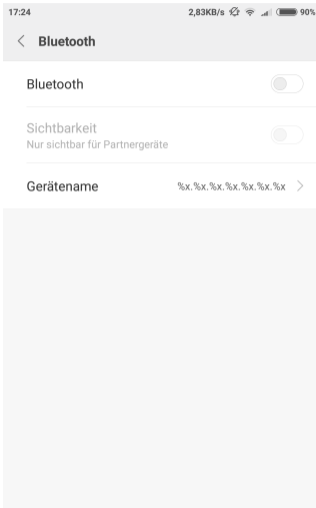


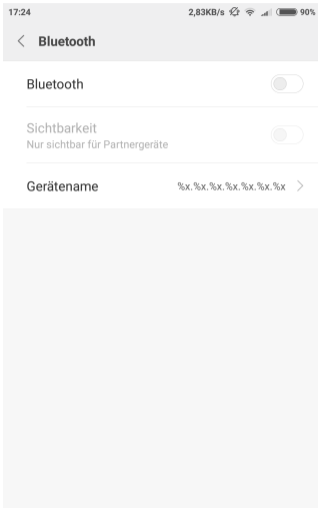
```
% valgrind ./format "ABCDABCD %p %p %p %p %p %p %p %n"  
[...]  
==17472== Invalid write of size 4  
==17472==      at 0x4E89533: vfprintf (vfprintf.c:1631)  
==17472==      by 0x4E8F898: printf (printf.c:33)  
==17472==      by 0x40061E: main (printf.c:6)  
==17472== Address 0x4443424144434241 is 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)







- 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



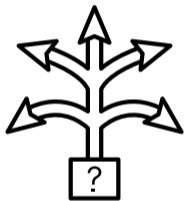
Find a format string to extract the binary's secret

- The binary:
<https://sasd.is.attacking.systems/challenges#format>
- 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

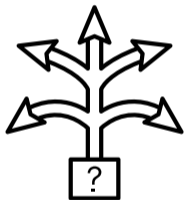
Source

```
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);
}
```

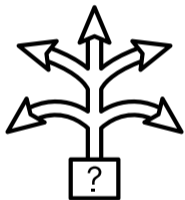
Type Confusion



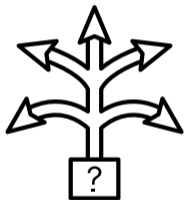
- A **resource** from one type is allocated, but later **referenced as a different type**
- No problem if the types are compatible (\Rightarrow C++ polymorphism)
- C/C++ also allows **casts to incompatible types**, leading to logic errors
- Accesses can be **out-of-bounds** (\Rightarrow buffer overflow),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



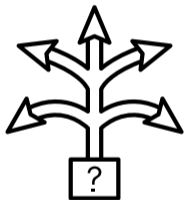
- A **resource** from one type is allocated, but later **referenced as a different type**
- No problem if the types are compatible (\Rightarrow C++ polymorphism)
- C/C++ also allows **casts to incompatible types**, leading to logic errors
- Accesses can be **out-of-bounds** (\Rightarrow buffer overflow),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



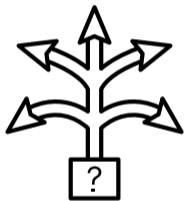
- A **resource** from one type is allocated, but later **referenced as a different type**
- No problem if the types are compatible (\Rightarrow C++ polymorphism)
- C/C++ also allows **casts to incompatible types**, leading to logic errors
- Accesses can be **out-of-bounds** (\Rightarrow buffer overflow),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



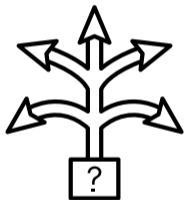
- A **resource** from one type is allocated, but later **referenced as a different type**
- No problem if the types are compatible (\Rightarrow C++ polymorphism)
- C/C++ also allows **casts to incompatible types**, leading to logic errors
- Accesses can be **out-of-bounds** (\Rightarrow buffer overflow),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



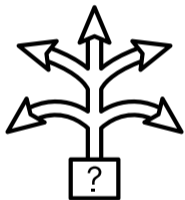
- C++ provides different types of casts
- `dynamic_cast`: Explicit type checks at runtime, but slow
- `static_cast`: Type check only at compile time, type confusion if runtime type is unexpected
- `reinterpret_cast`: Allows to explicitly break type checks



- 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



- 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



- 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



```
#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;
}
```




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



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



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



Practical Example Analysis: Type Confusion



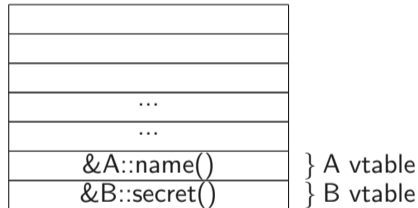
```
class A {
    public: virtual const char* name()
        { return "A"; };
};

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





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

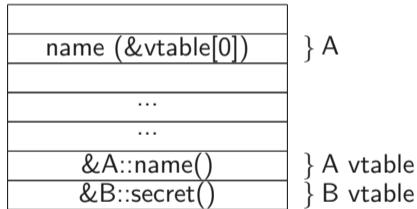
class B {
    public: const char* name ()
        { return "B"; };
    private: virtual const char* secret ()
        { return "secret"; };
};

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 →





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

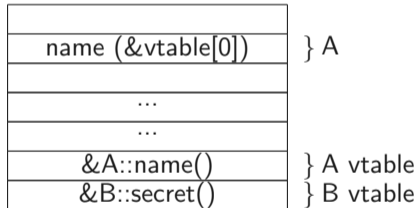
class B {
    public: const char* name()
        { return "B"; };
    private: virtual const char* secret()
        { return "secret"; };
};

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 →





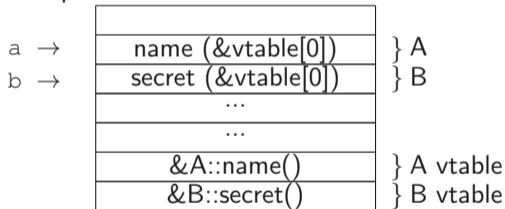
```
class A {
    public: virtual const char* name()
        { return "A"; };
};

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





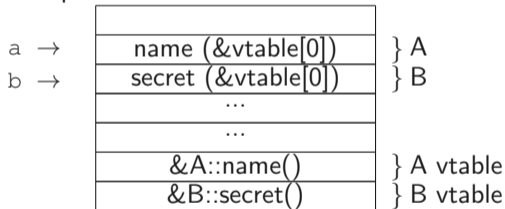
```
class A {
    public: virtual const char* name()
        { return "A"; };
};

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





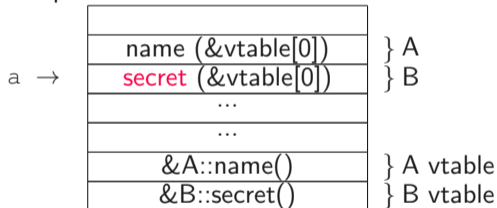
```
class A {
    public: virtual const char* name ()
        { return "A"; };
};

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





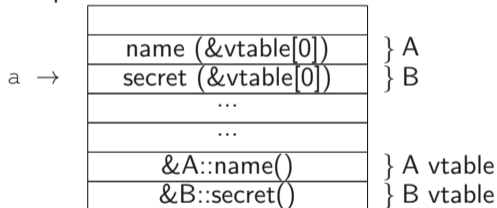
```
class A {
    public: virtual const char* name()
        { return "A"; };
};

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



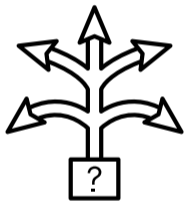


```
% ./g++ vtables.h -fdump-lang-class && cat vtables.h.0011.class
Vtable for A                                Vtable for B
A::_ZTV1A: 3 entries                          B::_ZTV1B: 3 entries
0      (int (*)(...))0                       0      (int (*)(...))0
8      (int (*)(...))(& _ZTI1A)              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



- A type confusion happens if a pointer (or object) is **casted** to a **wrong object**
- It allows to
 - **execute** (arbitray) code
 - read/write out-of-bounds
 - crash the application
- Relatively new type of memory corruption
- Impacts not thoroughly studied yet...

- Type confusion bugs were exploited in many applications
 - Adobe Flash (CVE-2015-3077)
 - Microsoft Internet Explorer (CVE-2015-6184)
 - PHP (CVE-2016-3185)
 - Google Chrome (CVE-2013-0912)
- 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

- Type confusion bugs were exploited in many applications
 - Adobe Flash (CVE-2015-3077)
 - Microsoft Internet Explorer (CVE-2015-6184)
 - PHP (CVE-2016-3185)
 - Google Chrome (CVE-2013-0912)
- 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

- Type confusion bugs were exploited in many applications
 - Adobe Flash (CVE-2015-3077)
 - Microsoft Internet Explorer (CVE-2015-6184)
 - PHP (CVE-2016-3185)
 - Google Chrome (CVE-2013-0912)
- Generally play an important role in **browser exploits**
- You can also be like Mozilla and combine them with other bugs:

A screenshot of a document with a dark red header and light gray body. The header contains the text 'Mozilla Foundation Security Advisory 2015-39'. Below the header, the text 'Use-after-free due to type confusion flaws' is visible, followed by a horizontal dotted line.

Mozilla Foundation Security
Advisory 2015-39

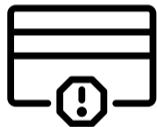
Use-after-free due to type confusion flaws

- Type confusion bugs were exploited in many applications
 - Adobe Flash (CVE-2015-3077)
 - Microsoft Internet Explorer (CVE-2015-6184)
 - PHP (CVE-2016-3185)
 - Google Chrome (CVE-2013-0912)
- Generally play an important role in **browser exploits**
- You can also be like Mozilla and combine them with other bugs:

A screenshot of a document with a dark red header and light gray body. The header contains the text 'Mozilla Foundation Security Advisory 2015-39'. Below the header, the text 'Use-after-free due to type confusion flaws' is visible, followed by a horizontal dotted line.

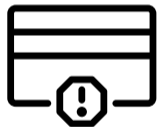
Mozilla Foundation Security
Advisory 2015-39

Use-after-free due to type confusion flaws



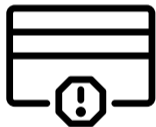
Invalid memory accesses...

- 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++



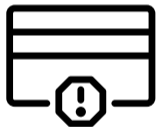
Invalid memory accesses...

- 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++



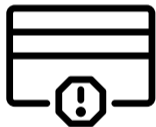
Invalid memory accesses...

- 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++



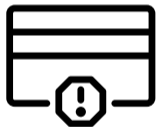
Invalid memory accesses...

- 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++



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++



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 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**



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**



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**



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**



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**



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**





- 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



- 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



- 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



- 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



- 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



- 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



- 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



- 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

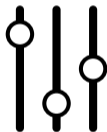


- Some bugs might not be exclusively in the binary
- They appear due to the program's interaction with the **environment**
 - Environment variables
 - Loader
 - Access control
- These factors have to be considered when writing programs

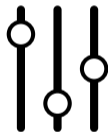


- Some bugs might not be exclusively in the binary
- They appear due to the program's interaction with the **environment**
 - Environment variables
 - Loader
 - Access control
- These factors have to be considered when writing programs

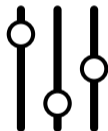




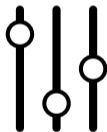
- **Named values** of the environment, usable by programs
- Each process has its **own set** (usually copy of the parent)
- Provided by the *envp* pointer of *exec*
- Can also be set using `VARIABLE=VALUE` in most shells
- Accessed using `setenv/getenv` in C/C++



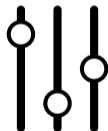
- **Named values** of the environment, usable by programs
- Each process has its **own set** (usually copy of the parent)
 - Provided by the *envp* pointer of *exec*
 - Can also be set using `VARIABLE=VALUE` in most shells
 - Accessed using `setenv/getenv` in C/C++



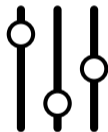
- **Named values** of the environment, usable by programs
- Each process has its **own set** (usually copy of the parent)
- Provided by the *envp* pointer of *exec*
 - Can also be set using `VARIABLE=VALUE` in most shells
 - Accessed using `setenv/getenv` in C/C++



- **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++



- **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++



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



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



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



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



Practical Example: Buffer Overflow



```
#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);
}
```



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

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

Program received signal SIGSEGV, Segmentation fault.
0x00000000040061d in greetings (hello=1) at envovf.c:12
(gdb) bt
#0  0x00000000040061d in greetings (hello=1) at envovf.c:12
#1  0x4141414141414141 in ?? ()
```



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

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

Program received signal SIGSEGV, Segmentation fault.
0x00000000040061d in greetings (hello=1) at envovf.c:12
(gdb) bt
#0  0x00000000040061d in greetings (hello=1) at envovf.c:12
#1  0x4141414141414141 in ?? ()
```



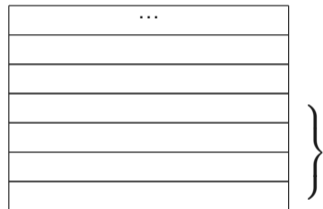

Practical Example Analysis: Buffer Overflow



```
#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);
}
```

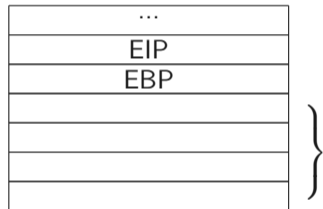




```
#include <stdio.h>
#include <stdlib.h>

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);
}
```

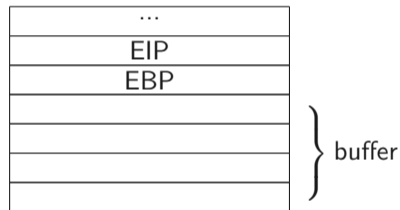




```
#include <stdio.h>
#include <stdlib.h>

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);
}
```

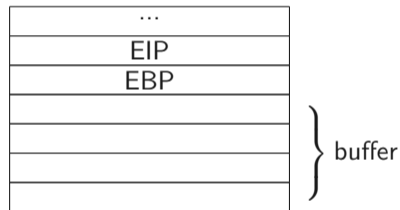




```
#include <stdio.h>
#include <stdlib.h>

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);
}
```

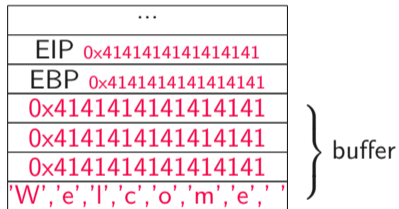




```
#include <stdio.h>
#include <stdlib.h>

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);
}
```





Practical Example Impact: Buffer Overflow



- 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



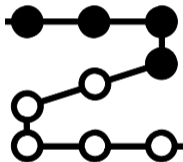
- Same impact as **classical stack buffer overflow**
- Attacker can jump to arbitrary location in memory
- Every function that is mapped in the address space can be executed
- Attacker has effectively **full control** over the program



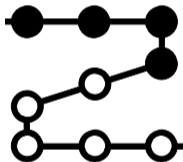
- Same impact as **classical stack buffer overflow**
- Attacker can jump to arbitrary location in memory
- Every function that is mapped in the address space can be executed
- Attacker has effectively **full control** over the program



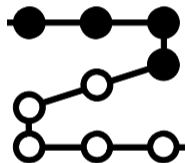
- Same impact as **classical stack buffer overflow**
- Attacker can jump to arbitrary location in memory
- Every function that is mapped in the address space can be executed
- Attacker has effectively **full control** over the program



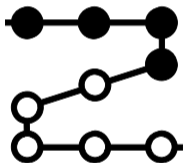
- If a binary to execute does not have full path (e.g. `/bin/lis`), folders in *PATH* variable are searched
- As soon as binary is found in one of these folders, it is executed
- Not only shell does that, but also `exec1p`, `execvp`, and `system`
- Attacker might prepend folder to *PATH* variable



- If a binary to execute does not have full path (e.g. `/bin/lis`), 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 `exec1p`, `execvp`, and `system`
- Attacker might prepend folder to *PATH* variable



- If a binary to execute does not have full path (e.g. `/bin/lis`), 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 `exec1p`, `execvp`, and `system`
- Attacker might prepend folder to *PATH* variable



- If a binary to execute does not have full path (e.g. `/bin/lis`), 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 `exec1p`, `execvp`, and `system`
- Attacker might prepend folder to *PATH* variable



Fun Example: PATH manipulation



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

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



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

```
% 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.
```



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

```
% 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 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



- 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



- 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



- 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



```
#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");
    }
}
```



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

```
#include <stdio.h>  
char *strcpy(char *dest, const char *src) {  
    printf("Copy: %s\n", src);  
    while((*dest++ = *src++));  
}
```

```
% gcc -shared -fPIC strcpy.c -o strcpy.so  
% LD_PRELOAD=$PWD/strcpy.so ./secret  
Copy: ultra secret password  
Only root can get the password
```



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

```
#include <stdio.h>  
char *strcpy(char *dest, const char *src) {  
    printf("Copy: %s\n", src);  
    while((*dest++ = *src++));  
}
```

```
% gcc -shared -fPIC strcpy.c -o strcpy.so  
% LD_PRELOAD=$PWD/strcpy.so ./secret  
Copy: ultra secret password  
Only root can get the password
```



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

```
#include <stdio.h>  
char *strcpy(char *dest, const char *src) {  
    printf("Copy: %s\n", src);  
    while((*dest++ = *src++));  
}
```

```
% gcc -shared -fPIC strcpy.c -o strcpy.so  
% LD_PRELOAD=$PWD/strcpy.so ./secret  
Copy: ultra secret password  
Only root can get the password
```

Live Demo

Cheating in Tetris with LD_PRELOAD





- 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 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 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 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: 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: 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: 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: 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**



- 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**



- 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**



- 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**



- 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



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

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



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

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



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

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



- 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



- 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



- 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



- 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



```
#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:::
```



```
#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:::
```



```
#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:::
```



```
#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:::
```




```
#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:::
```



```
#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:::
```



```
#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:::
```



```
#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:::
```



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



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



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







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



Questions?

© 1999 Randy Glasbergen. www.glasbergen.com



**"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."**

-  Jonathan Afek and Adi Sharabani.
Dangling pointer: Smashing the pointer for fun and profit.
Black Hat USA, 2007.
-  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.
-  Natalie Silvanovich, Dazed, (Type) Confused.
One perfect bug: Exploiting type confusion in flash.
-  scut / team teso.
Exploiting format string vulnerabilities.

-  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.