Software Security 03

Sven Schäge

Learning Goals

- Understand Memory Corruption Attacks
  - And why they are unlikely to vanish soon
- Understand code injection in particular and reason under what circumstances the attack works
- Deeply understand pointers and the dangers that come with using them
- Derive countermeasures and argue about benefits and drawbacks

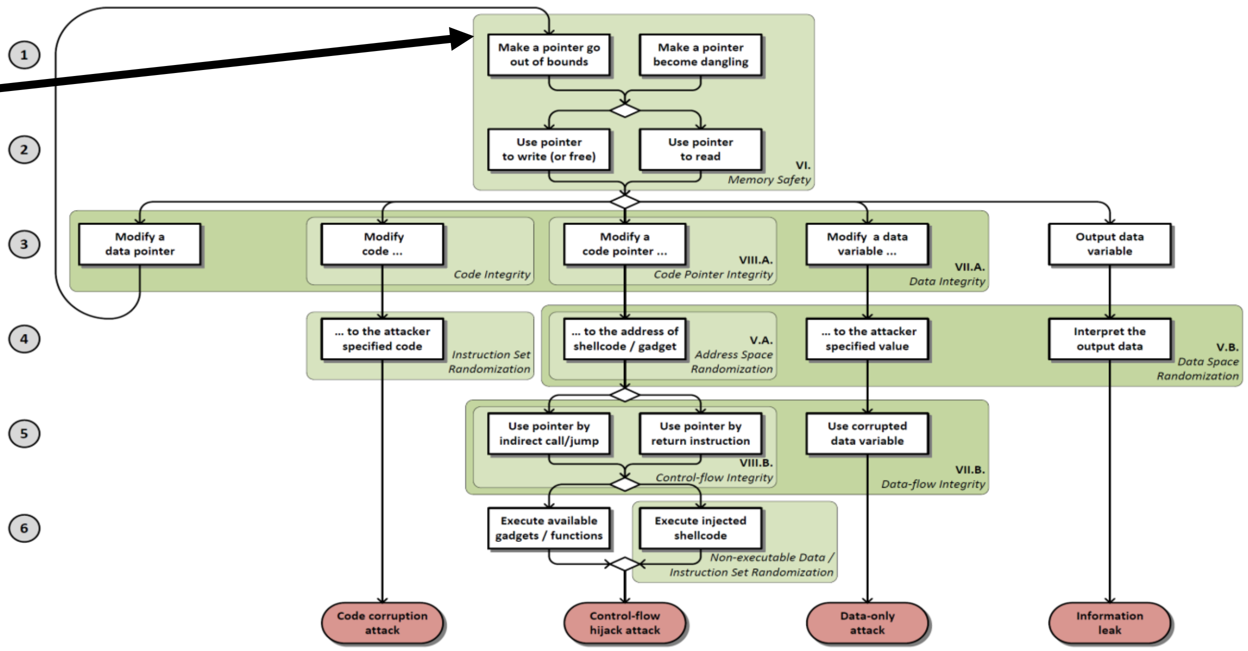
Introduction to Memory Corruption Attacks

Sven Schäge

Memory Corruption Attacks/Buffer Overflows

- Buffer Overflows are a special form of memory corruption attacks.
- Code written in C/C++ is very susceptible

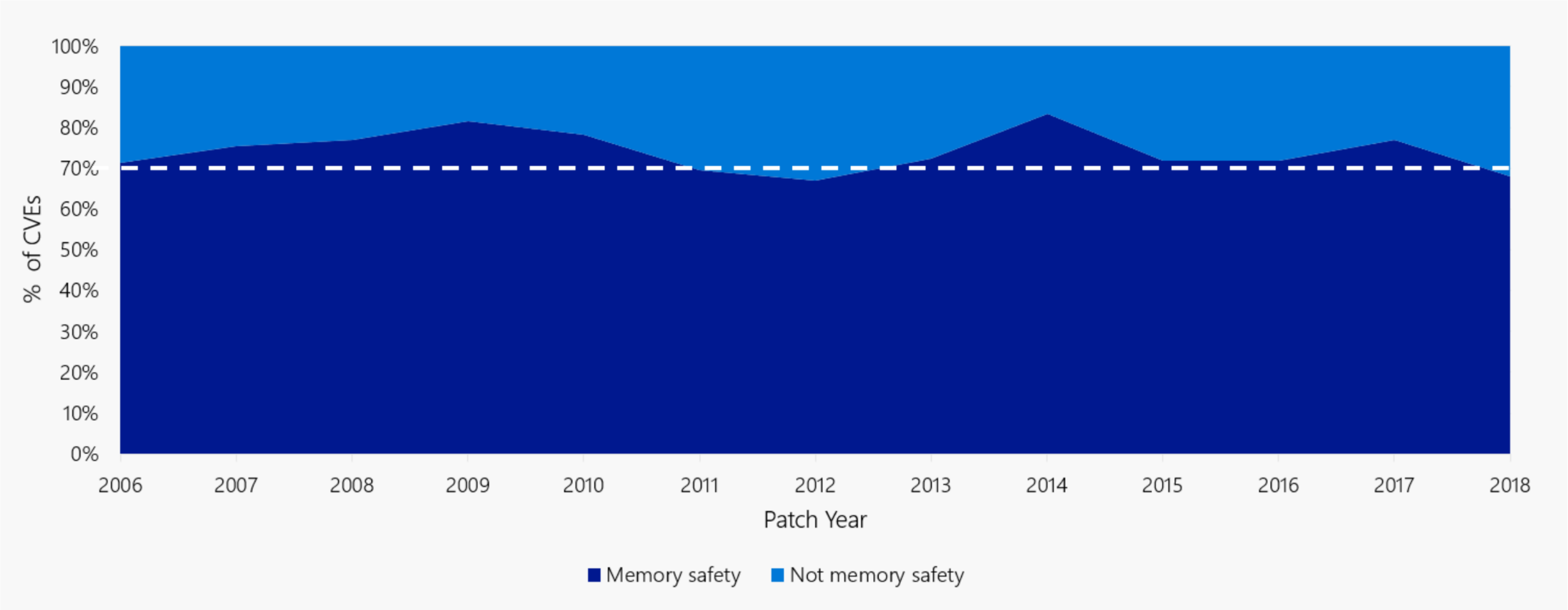
buffer overflow
(narrower sense)



buffer overflow
=memory corruption attacks
(wider sense)

Source:
SoK: Eternal War in Memory: Szekeres, Payer, Wei, Song
Oakland 2013

Importance of Memory Corruption Attacks



Memory corruption vs non-memory corruption bugs at Microsoft 2006-2018

https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/

Example Program: Simple Password Checker (SPC)

C code

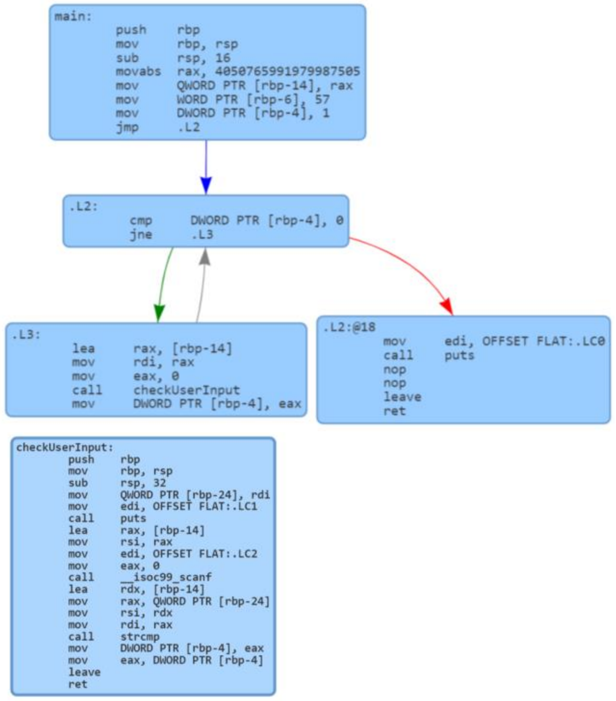
```c
#include <stdio.h>
int main() //function to authenticate Alice, asks for password
{
    char pwd[10]="123456789"; //Alice's password, hardcoded
    int j=1; //j signals if input==pwd (case j==0), initialized to inequal

    while (j!=0) //loop only left if user input==pwd
    {
        j=checkUserInput(pwd);
    }

    printf("Your input is valid. Go on!\n");
    return 0; //only halts if correctly authenticated

}

int checkUserInput(char toCompare[10])
{
    char userInput[10];
    int i;

    printf("Enter your Password!\n");
    scanf("%s", userInput);

    i=strcmp(toCompare,userInput);//outputs 0 only if equal!

    return i;
}
```

SPC

Assembler code

```
.LC0:
        .string "Your input is valid. Go on!"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        movabs  rax, 4050765991979987505
        mov     QWORD PTR [rbp-14], rax
        mov     WORD PTR [rbp-6], 57
        mov     DWORD PTR [rbp-4], 1
        jmp     .L2
.L3:
        lea     rax, [rbp-14]
        mov     rdi, rax
        mov     eax, 0
        call    checkUserInput
        mov     DWORD PTR [rbp-4], eax
.L2:
        cmp     DWORD PTR [rbp-4], 0
        jne     .L3
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        mov     eax, 0
        leave
        ret
.LC1:
        .string "Enter your Password!"
.LC2:
        .string "%s"
checkUserInput:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     QWORD PTR [rbp-24], rdi
        mov     edi, OFFSET FLAT:.LC1
        call    puts
        lea     rax, [rbp-14]
        mov     rsi, rax
        mov     edi, OFFSET FLAT:.LC2
        mov     eax, 0
        call    __isoc99_scanf
        lea     rdx, [rbp-14]
        mov     rax, QWORD PTR [rbp-24]
        mov     rsi, rdx
        mov     rdi, rax
        call    strcmp
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        leave
        ret
```

Graph output



C code

```c
#include <stdio.h>
int main() //function to authenticate Alice, asks for password
{
    char pwd[10]="123456789"; //Alice's password, hardcoded
    int j=1; //j signals if input==pwd (case j==0), initialized to inequal

    while (j!=0) //loop only left if user input==pwd
    {
        j=checkUserInput(pwd);
    }

    printf("Your input is valid. Go on!\n");
    return 0; //only halts if correctly authenticated

}

int checkUserInput(char toCompare[10])
{
    char userInput[10];
    int i;

    printf("Enter your Password!\n");
    scanf("%s", userInput);

    i=strcmp(toCompare,userInput);//outputs 0 only if equal!

    return i;
}
```

SPC

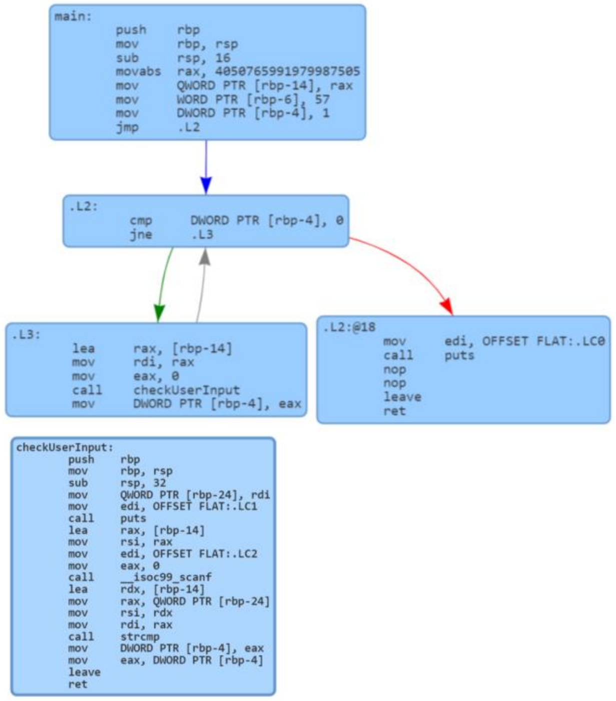
C code

```c
#include <stdio.h>
int main() //function to authenticate Alice, asks for password
{
    char pwd[10]="123456789"; //Alice's password, hardcoded
    int j=1; //j signals if input==pwd (case j==0), initialized to inequal

    while (j!=0) //loop only left if user input==pwd
    {
        j=checkUserInput(pwd);
    }

    printf("Your input is valid. Go on!\n");
    return 0; //only halts if correctly authenticated

}

int checkUserInput(char toCompare[10])
{
    char userInput[10];
    int i;

    printf("Enter your Password!\n");
    scanf("%s", userInput);

    i=strcmp(toCompare,userInput);//outputs 0 only if equal!

    return i;
}
```

Assembler code

```
.LC0:
        .string "Your input is valid. Go on!"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        movabs  rax, 4050765991979987505
        mov     QWORD PTR [rbp-14], rax
        mov     WORD PTR [rbp-6], 57
        mov     DWORD PTR [rbp-4], 1
        jmp     .L2
.L3:
        lea     rax, [rbp-14]
        mov     rdi, rax
        mov     eax, 0
        call    checkUserInput
        mov     DWORD PTR [rbp-4], eax
.L2:
        cmp     DWORD PTR [rbp-4], 0
        jne     .L3
        mov     edi, OFFSET FLAT:.LC0
        call    puts
        mov     eax, 0
        leave
        ret
.LC1:
        .string "Enter your Password!"
.LC2:
        .string "%s"
checkUserInput:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 32
        mov     QWORD PTR [rbp-24], rdi
        mov     edi, OFFSET FLAT:.LC1
        call    puts
        lea     rax, [rbp-14]
        mov     rsi, rax
        mov     edi, OFFSET FLAT:.LC2
        mov     eax, 0
        call    __isoc99_scanf
        lea     rdx, [rbp-14]
        mov     rax, QWORD PTR [rbp-24]
        mov     rsi, rdx
        mov     rdi, rax
        call    strcmp
        mov     DWORD PTR [rbp-4], eax
        mov     eax, DWORD PTR [rbp-4]
        leave
        ret
```

Graph output

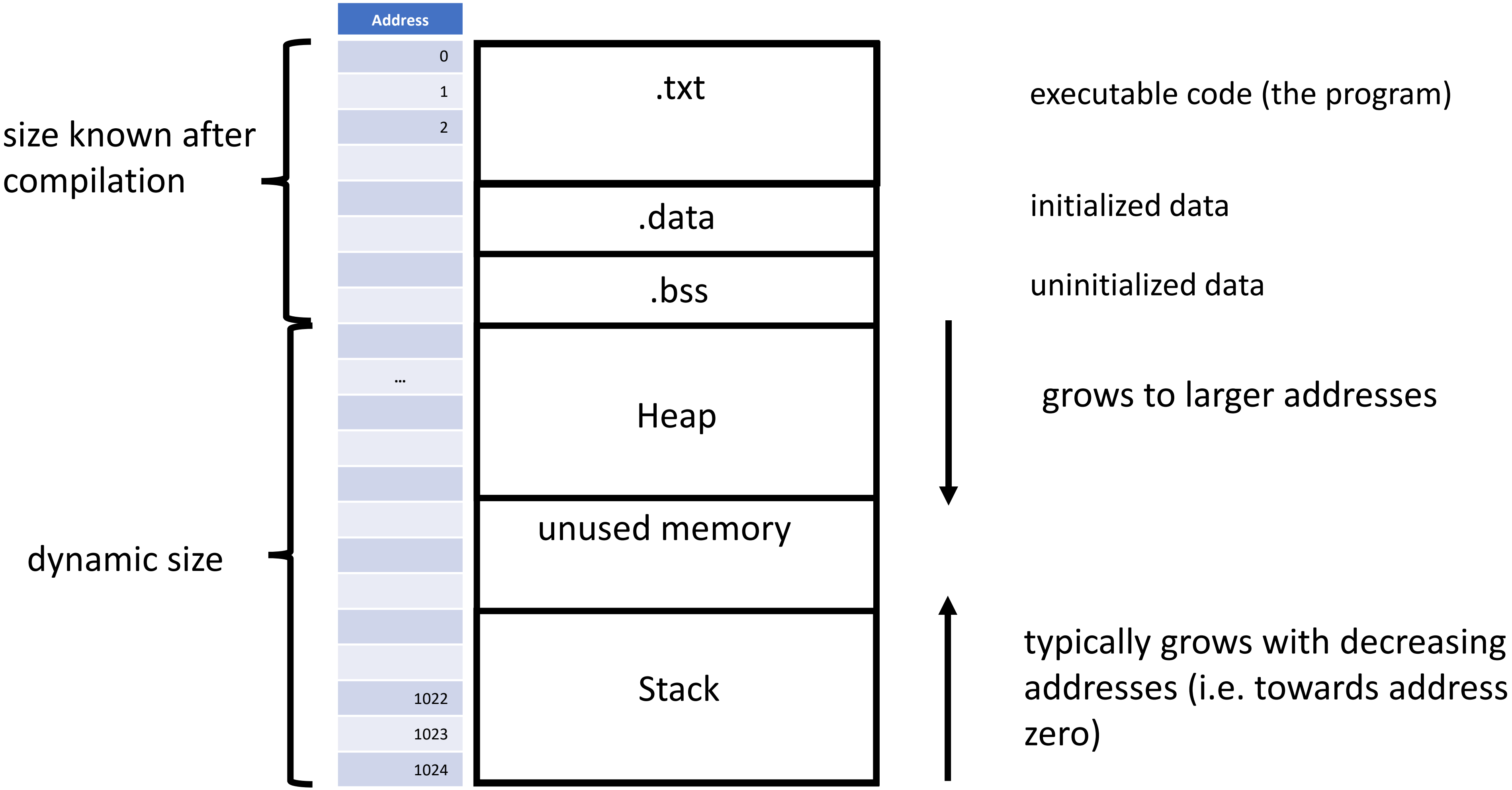



```
123456789

Program returned: 0
Program stdout
Enter your Password!
Your input is valid. Go on!
```

```
12345678

Program returned: 143
Program stdout
Enter your Password!
Enter your Password!
Enter your Password!
```

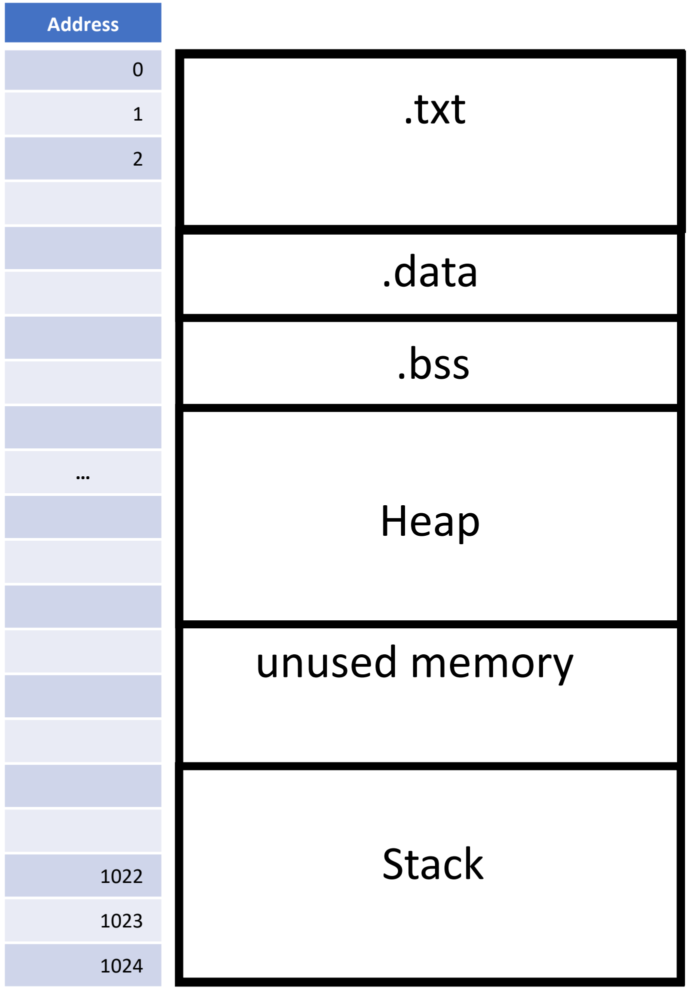
Recall: Program Memory Layout

Random Access Memory



Recall: Program Memory Layout

Random Access Memory



Stack used to manage call to checkUserInput!
-holds return address
-holds local variables

```
char userInput[10];
```

Recall: Program Memory Layout

Random Access Memory

```c
#include <stdio.h>
int main() //function to authenticate Alice, asks for password
{
    char pwd[10]="123456789"; //Alice's password, hardcoded
    int j=1; //j signals if input==pwd (case j==0), initialized to inequal

    while (j!=0) //loop only left if user input==pwd
    {
        j=checkUserInput(pwd);
    }

    printf("Your input is valid. Go on!\n");
    return 0; //only halts if correctly authenticated

}

int checkUserInput(char toCompare[10])
{
    char userInput[10];
    int i;

    printf("Enter your Password!\n");
    scanf("%s", userInput);

    i=strcmp(toCompare,userInput);//outputs 0 only if equal!

    return i;
}
```

Stack used to manage call to `checkUserInput` and return to `main`
-holds return address
-holds local variables including:

`char userInput[10];`

Address
0
1
2
…
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024

Stack

```
userInput[0]
userInput[1]
userInput[2]
userInput[3]
userInput[4]
userInput[5]
userInput[6]
userInput[7]
userInput[8]
userInput[9]
```

`char userInput[10];`

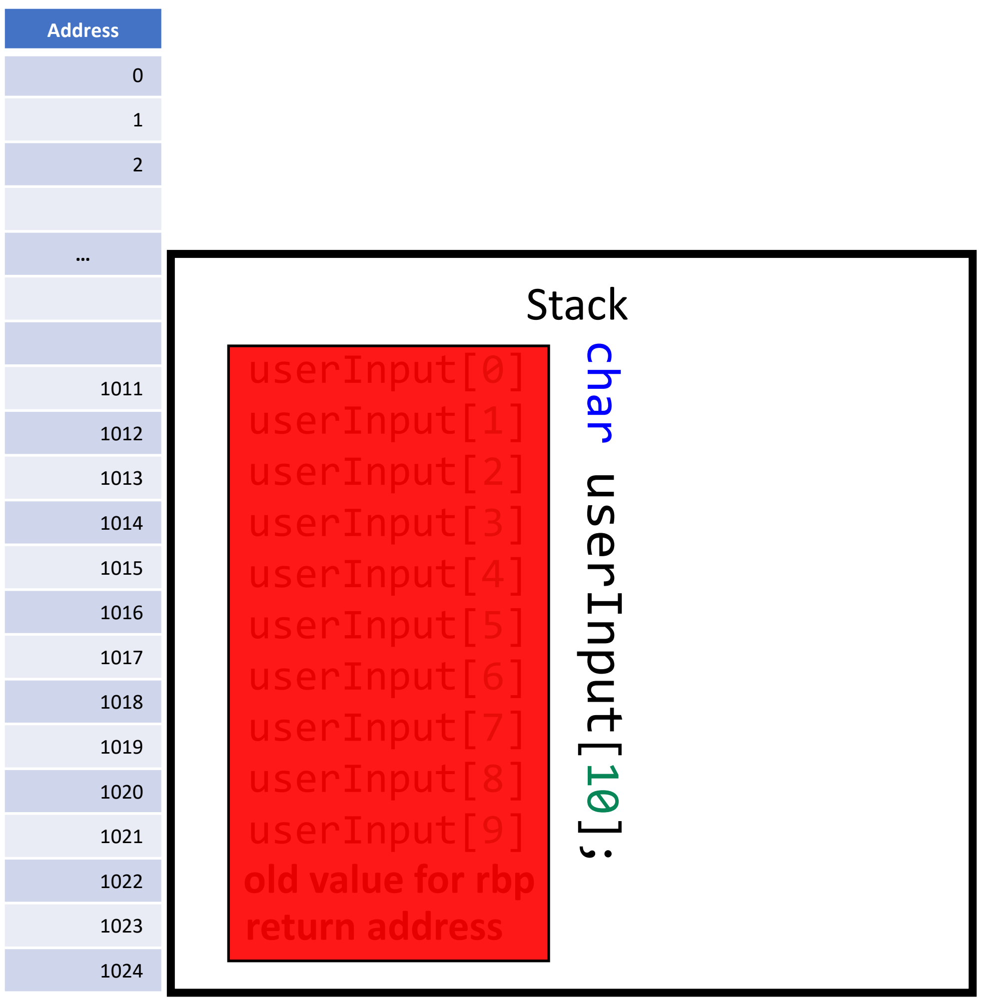
old value of bp
return address

Value	Address	Comment
	[ebp – X]	current stack pointer
	…	
	[ebp – 8]	2nd local variable
	[ebp – 4]	1st local variable
oldbp	[ebp]	old base pointer
RA of caller	[ebp + 4]	
10	[ebp + 8]	1st function argument
5	[ebp + 12]	2nd function argument
2	[ebp + 16]	3rd function argument
	…	

Simple Buffer Overflow (Narrower Sense)

```c
#include <stdio.h>
int main() //function to authenticate Alice, asks for password
{
    char pwd[10]="123456789"; //Alice's password, hardcoded
    int j=1; //j signals if input==pwd (case j==0), initialized to inequal

    while (j!=0) //loop only left if user input==pwd
    {
        j=checkUserInput(pwd);
    }

    printf("Your input is valid. Go on!\n");
    return 0; //only halts if correctly authenticated

}

int checkUserInput(char toCompare[10])
{
    char userInput[10];
    int i;

    printf("Enter your Password!\n");
    scanf("%s", userInput);

    i=strcmp(toCompare,userInput);//outputs 0 only if equal!

    return i;
}
```

Random Access Memory



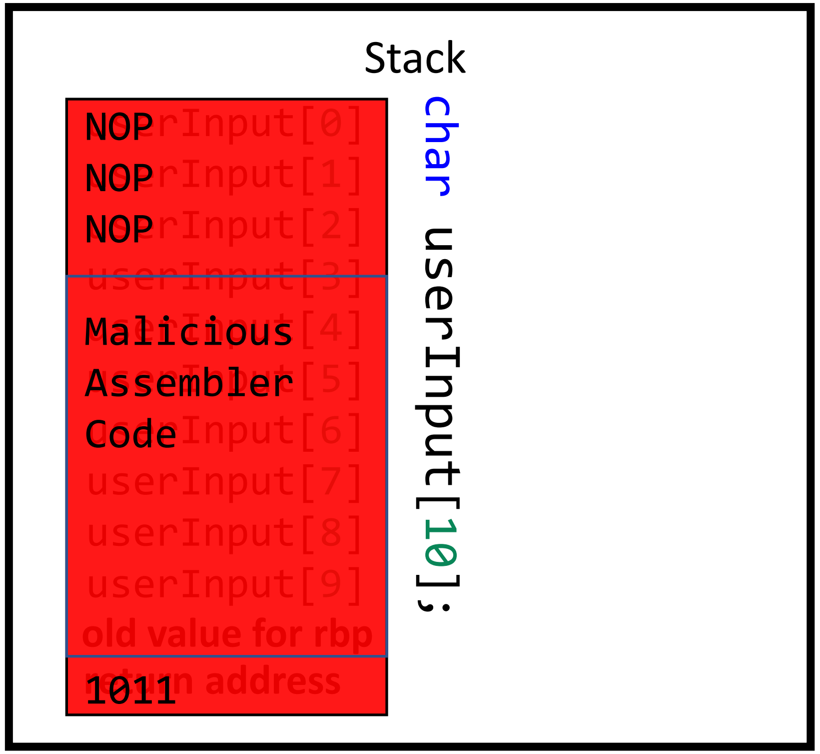
The attacker
- inputs more characters than the variables are supposed to hold
- overwrites return address

Simple Buffer Overflow Attack and Exploit

Random Access Memory

```c
#include <stdio.h>
int main() //function to authenticate Alice, asks for password
{
    char pwd[10]="123456789"; //Alice's password, hardcoded
    int j=1; //j signals if input==pwd (case j==0), initialized to inequal

    while (j!=0) //loop only left if user input==pwd
    {
        j=checkUserInput(pwd);
    }

    printf("Your input is valid. Go on!\n");
    return 0; //only halts if correctly authenticated

}

int checkUserInput(char toCompare[10])
{
    char userInput[10];
    int i;

    printf("Enter your Password!\n");
    scanf("%s", userInput);

    i=strcmp(toCompare,userInput);//outputs 0 only if equal!

    return i;
}
```

Address
0
1
2
…
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024



The attacker

- inputs more characters than the variables are supposed to hold
- inserts malicious assembler code into stack
- overwrites return address to point to malicious code

Exploitation

Overview Exploitation

- Code Injection
  - Insert code to be executed directly as input
  - Challenge: write code without nullbytes which would indicate input end
- Arc Injection
  - Jump to other position of program
    - e.g. branch that would be executed in case password was verified
- Return-to-libc
  - Call functions present in a standard library that preexists in memory (libc)
  - Building blocks: functions in libc. (Can be easily deleted without hurting functionality of libc too much.)

Overview Exploitation

- Return-oriented Programming
  - Jump to code positions in independent library code which preexists in memory
  - The jump is to a location that is ended by a return (ret) statement (not necessarily the start of a function definition)
  - These jumps may be used to perform specific operations for the attacker. They are called gadgets.
  - For x86, a Turing complete set of gadgets is known.
  - This means that just via jumps to preexisting code, the attacker can execute any software it wants.
  - Building blocks: small code snippets in libc. (May consist of bits that are originally code or data. Harder to eliminate without destroying functionality of libc. x86 has “dense geometry”!)

Two instructions in the entrypoint `ecb_crypt` are encoded as follows:

```
f7 c7 07 00 00 00        test $0x00000007, %edi
0f 95 45 c3              setnzb -61(%ebp)
```

Starting one byte later, the attacker instead obtains

```
c7 07 00 00 00 0f        movl $0x0f000000, (%edi)
95                       xchg %ebp, %eax
45                       inc %ebp
c3                       ret
```

Source:
Hovav Shacham: The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86), CCS‘07

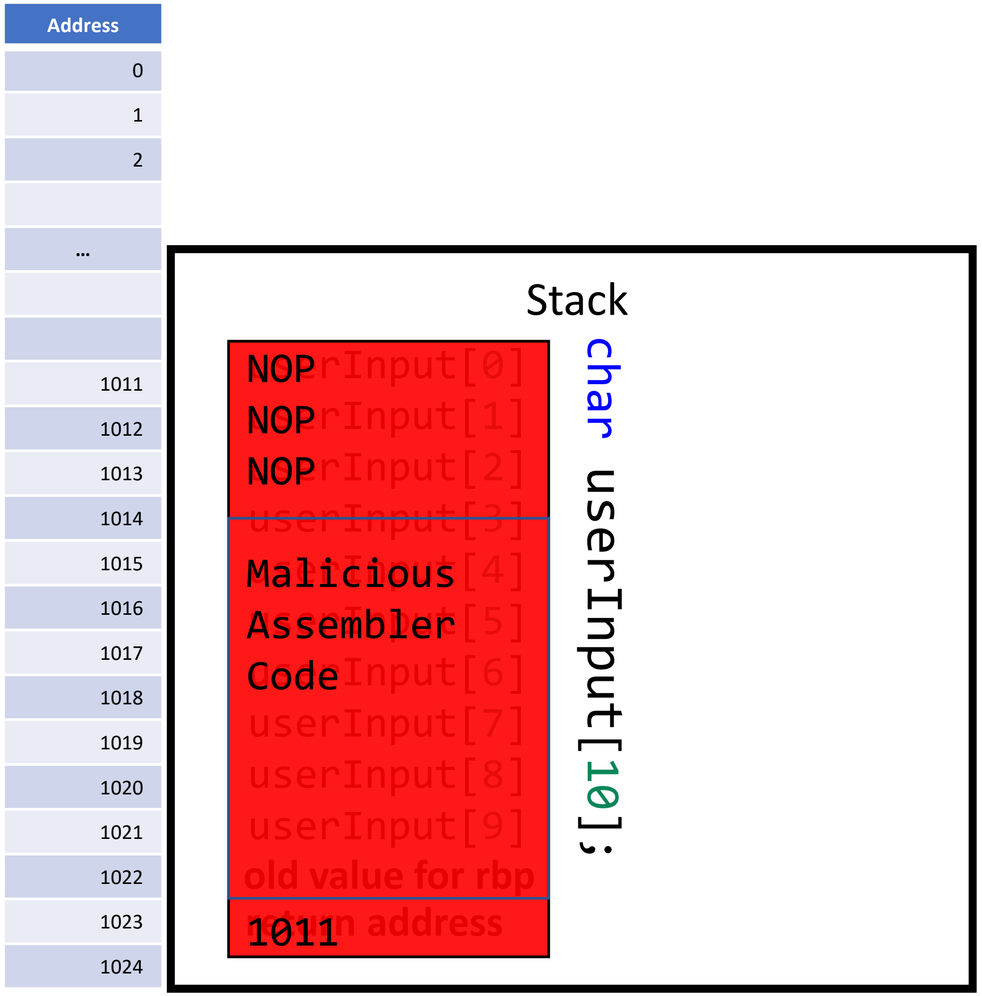
Buffer Overflow Attacks - Prerequisites and Countermeasures

Sven Schäge

Recall: Simple Buffer Overflow Attack and Exploit

Random Access Memory

```c
#include <stdio.h>
int main() //function to authenticate Alice, asks for password
{
    char pwd[10]="123456789"; //Alice's password, hardcoded
    int j=1; //j signals if input==pwd (case j==0), initialized to inequal

    while (j!=0) //loop only left if user input==pwd
    {
        j=checkUserInput(pwd);
    }

    printf("Your input is valid. Go on!\n");
    return 0; //only halts if correctly authenticated

}

int checkUserInput(char toCompare[10])
{
    char userInput[10];
    int i;

    printf("Enter your Password!\n");
    scanf("%s", userInput);

    i=strcmp(toCompare,userInput);//outputs 0 only if equal!

    return i;
}
```

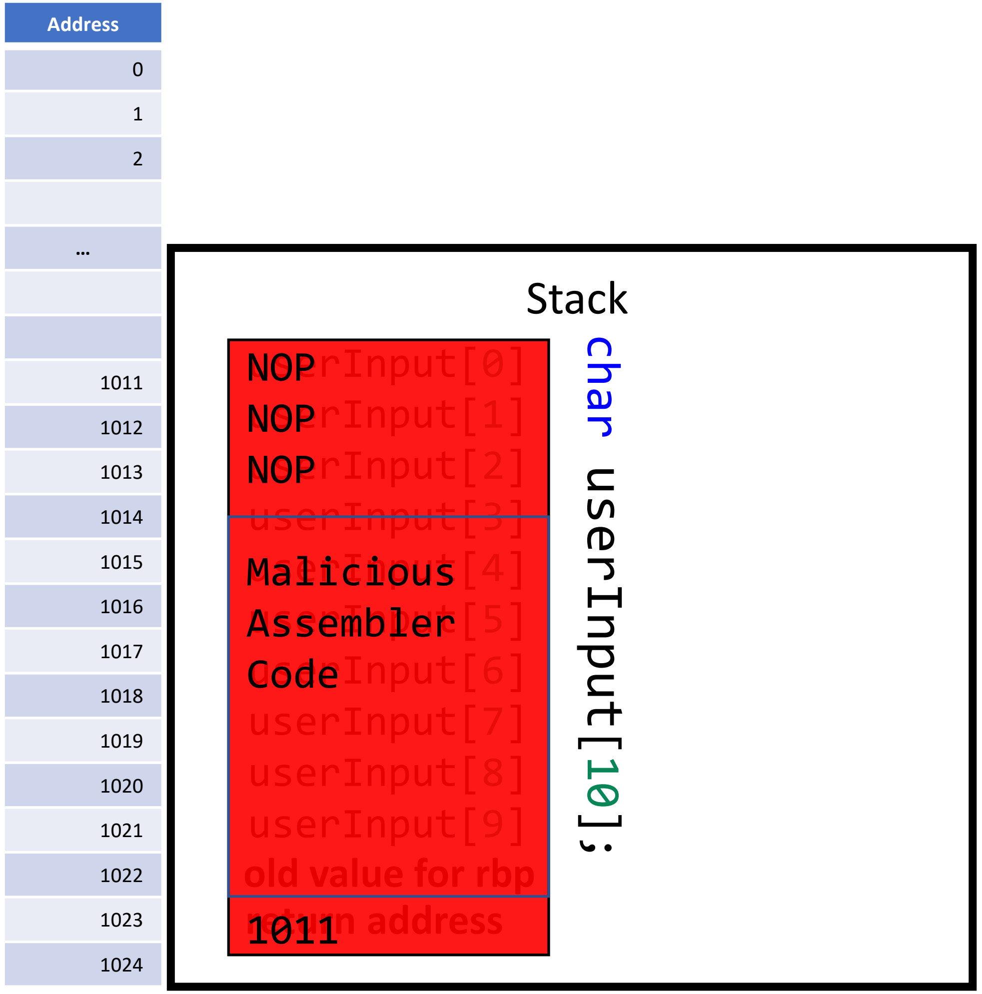


The attacker
- inputs more characters than the variables are supposed to hold
- inserts malicious assembler code into stack
- overwrites return address to point to malicious code

Recall: Simple Buffer Overflow Attack and Exploit

Random Access Memory

```c
#include <stdio.h>
int main() //function to authenticate Alice, asks for password
{
    char pwd[10]="123456789"; //Alice's password, hardcoded
    int j=1; //j signals if input==pwd (case j==0), initialized to inequal

    while (j!=0) //loop only left if user input==pwd
    {
        j=checkUserInput(pwd);
    }

    printf("Your input is valid. Go on!\n");
    return 0; //only halts if correctly authenticated

}

int checkUserInput(char toCompare[10])
{
    char userInput[10];
    int i;

    printf("Enter your Password!\n");
    scanf("%s", userInput);

    i=strcmp(toCompare,userInput);//outputs 0 only if equal!

    return i;
}
```



4 Observations

1. Buffer overflow is possible
2. Overwriting of return address does not raise alarms
3. Malicious code must be executable
4. Return address must map to malicous code

Countermeasures

- 4 Observations
  1. Buffer overflow is possible
  2. Overwriting of return address does not raise alarms
  3. Malicious code must be executable
  4. Return address must map to malicous code
- Countermeasures focus on these observations
  1. Range checks for variables;
     - Use type-safe memory language=>slower
     - Use fat pointers, which additionally hold information on the allowed range
  2. Implement check for integrity of return address, avoid illegal memory access
     - Stack canaries
  3. Make malicious code not executable, avoid illegal control flow
     - Read-only and write-only memory organization, DEP/W$\oplus$ X
  4. Make address organization unpredictable for attacker via randomization
     - Address Space Layout Randomization (ASLR)

costs: additional bookkeeping and runtime checks

Fat Pointers

- In programming languages like C/C++ pointers map to a starting location in memory
  - Subsequent addresses hold the data
  - Consider char pwd[10]=‘‘123456789‘‘;
  - Here pwd points to the address that starts to hold the password data in memory, a contionous sequence of memory units each holding a single character
- Fat pointers
  - additionally store the supposed length of the memory and
  - check at runtime if bounds are legal

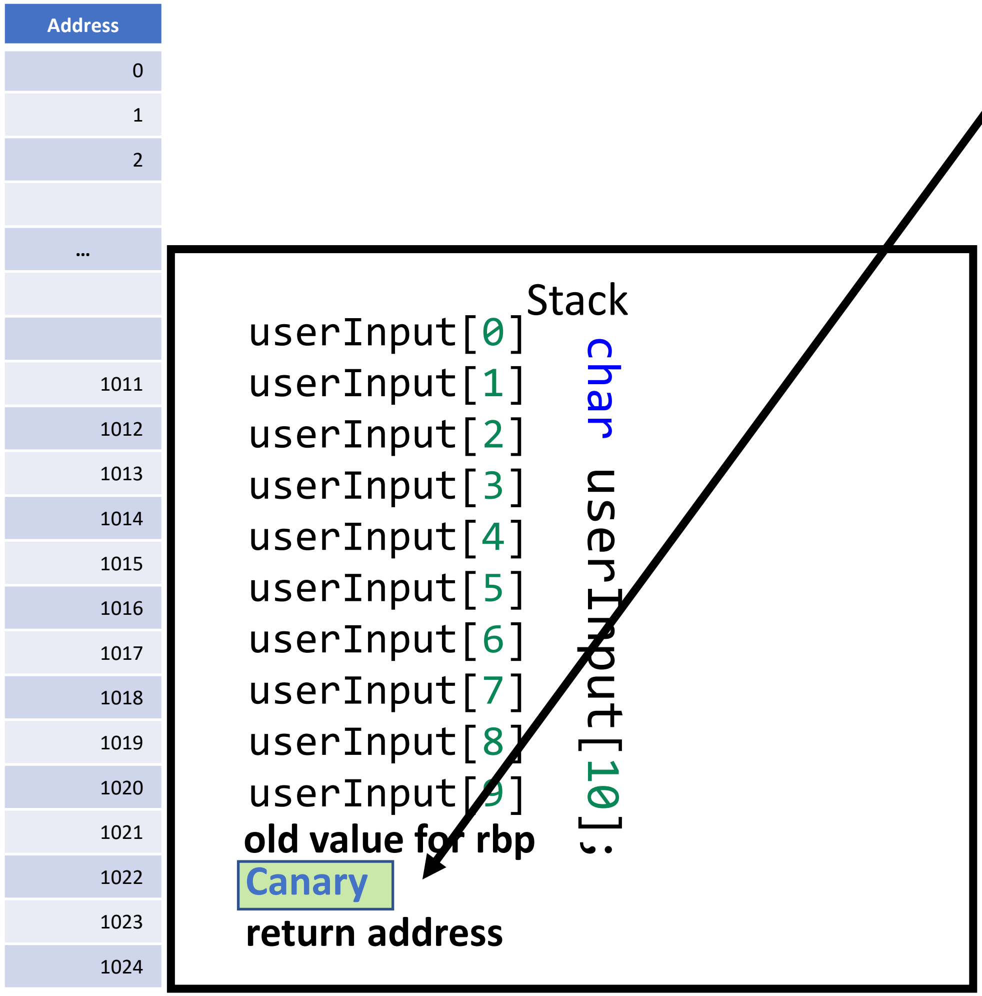
Limitations

- Fat pointers and type-safe languages decrease the efficiency since checks have to be performed at run-time
- However, they greatly increase security!

Stack Canaries

```c
#include <stdio.h>
int main() //function to authenticate Alice, asks for password
{
    char pwd[10]="123456789"; //Alice's password, hardcoded
    int j=1; //j signals if input==pwd (case j==0), initialized to inequal

    while (j!=0) //loop only left if user input==pwd
    {
        j=checkUserInput(pwd);
    }

    printf("Your input is valid. Go on!\n");
    return 0; //only halts if correctly authenticated

}

int checkUserInput(char toCompare[10])
{
    char userInput[10];
    int i;

    printf("Enter your Password!\n");
    scanf("%s", userInput);

    i=strcmp(toCompare,userInput);//outputs 0 only if equal!

    return i;
}
```

Random Access Memory



random value whose integrity will be checked before jumping return address

buffer overflow attacks will likely modify the canary

Limitations

- Read from stack
  - The attacker may insert code that step-wisely outputs contents of the stack
    - Reading from stack without modification POP can be done via a move command
  - Among the information, there is the value of the stack canary
  - With that knowledge the attacker may overwrite the return address without modifying the stack canary

Data Execution Prevention(DEP) or W$\bigoplus$ X

- Most of the code that is executed in a binary is in the .txt section
- Our simple buffer overflow attack writes additional code where pure data is expected
- Idea of DEP/W$\bigoplus$ X: (let the operating system) mark some or all writable regions of memory as non-executable
  - can prevent stack and heap memory areas from being executable

Limitations

- In just-in-time (JIT) compilation machine code is generated from some intermediate language (often called byte code) at run time
  - This code needs to be both written and executed and thus this memory is generally exempted from DEP/W$\oplus$ X protection
  - JIT compilers can be used to write malicious code to circumvent DEP/W$\oplus$ X
- Return-oriented programming, return to lib-c
  - Via a smart choice of return address, an attacker can make the control flow land in a memory region of a function (in some library) that is already present in memory and deemed executable
  - This can provide a small, useful functionality for the attacker (gadget)
  - By combining calls to gadgets, the attacker can obtain a Turing complete language – i.e. all functionalities that would be available via direct assembler instructions are available via gadgets

Address Space Layout Randomization

- Idea: when the program starts, arrange memory positions of important memory areas randomly like stack, heap, code, and libraries
- Attackers do not know where the areas are and are less likely to choose a good return address
  - Find a library position
  - Find the stack
- Wrong guesses result in a crash of the program so previous attempts do not help to succeed in current attempt, since the memory layout has a new randomization

Limitations

- For inserted code the attacker may use long nop sequences in front of the actual malicious code
  - This increases the chance of executing that code even if the overwritten return address does not precisely map to the start of the malicious code
- ASLR addresses could be leaked, ongoing research!
  - e.g. side-channel attack on CPU branch target prediction buffer

Jump Over ASLR:
Attacking Branch Predictors to Bypass ASLR

Dmitry Evtyushkin
Department of Computer Science
State University of New York
at Binghamton
devtyushkin@cs.binghamton.edu

Dmitry Ponomarev
Department of Computer Science
State University of New York
at Binghamton
dima@cs.binghamton.edu

Nael Abu-Ghazaleh
Computer Science and
Engineering Department
University of California, Riverside
naelag@ucr.edu

Memory Corruption: Other Security Vulnerabilities and Problems

Sven Schäge

Heap Overflow

Recall: Heap Memory in C/C++

- It is often not clear how much memory a program will require. This can be highly dependent on the input data.
  - Example: Think of a list structure that per character, generates a new list object.
- Reserving (allocating) the worst-case amount of memory at start-up is usually too wasteful.
- To obtain the required memory dynamically for new objects, a program can explicitly ask for a new set of consecutive memory units to be reserved on the heap memory (in C/C++ via the **malloc** or **new** command).
- In C/C++, this memory area is **referenced** by a **pointer**, (i.e. a mere address value that points to the first byte of the reserved memory=holds the address of the first byte of the reserved memory.)
- Moreover, C/C++ has means to instruct the processor to **dereference** the pointer, i.e. to evaluate the address and gain access to the memory units it points to.
- After usage, we have to tell the compiler to **free** the reserved memory again, so that it can be used for other purposes.
- It is common practice that a function which is supposed to return a pointer, does return a default pointer called **NULL pointer**, in case it encountered an error.

Buffer Overflow
Heap

```
struct BankAccount
{
    int PIN[4];
    int balance;
    int username[40];
};
```

Buffer Overflow
Corrupting vtables (virtual tables)

```
// Type your code here, or load an example.

//C++ only: late binding
//which move function exactly is evaluated
//will be determined at runtime dependent on
//type of the object

//objects
//each of them has associated to itself
//a unique way of moving
Rocket r;
Ship s;
Oil Tanker o;

//move each object
r.move();
s.move();
o.move();
```

- Which function is used exactly, is determined only at runtime.
- To this end the C++ runtime environment manages where pointers to objects are mapped to pointers of functions.

Problems that Make Working with Pointers Error-Prone

Lifecycle of a Pointer in C

- int *p=(int *)malloc(sizeof(int) * elements_wanted);
  - memory allocation in C
  - elements_wanted can be decided on at runtime
- Read and write memory pointed to by p
  - e.g. to store some (temporary) values
  - use pointer arithmetic to modify p to point to all the memory units allocated
  - dereference p to get access to the contents at that memory units
- free(p)
  - Allows memory manager to re-use that memory if needed
- p=NULL;
  - Makes sure pointer does not point to old memory units
  - NULL is a dedicated address that pointers cannot access

Typical Security-Critical Problems of Pointers

- In C the programmer is responsible to appropriately take care of pointers.
- NULL dereference
  - A pointer is assumed to map to some valid address although in fact it maps to the NULL pointer
- Dangling Pointer
  - A pointer maps to some memory adress that has already been freed
- Use-after-free or double free
  - A pointer is used after it has been freed
- Memory leak
  - Allocated memory is never freed

Strings and Arrays are Tricky and May Lead to Memory Corruption

- In C strings are typically represented as arrays of characters that are terminated by a termination symbol \0.
- Variable length inputs are read until the termination symbol is found
- int char[10];
  - Reserves 10 bytes starting with char[0],…,char[9]
  - At position char[9] we should have a termination symbol
- One-Off errors describe scenarios where the bounds are confused. This can
  - Allocate memory badly (too few memory)
  - Run over bounds of arrays

Integers May Wrap Around if Too Large and May Lead to Memory Corruption

- Integers can wrap around if too large
  - Tricky in pointer arithmetic
- Signed integers do not have a defined behaviour. It depends on the implementations (sign and magnitude vs. one‘s complement vs. two‘s complement.)
  - May wrap around or not
  - Compilers use freedom unspecified behaviour for optimization
- Unexpected wrap arounds can
  - Allocate memory badly (too much, too few memory)
  - Run over bounds of arrays