# Boundary Errors and Control Hijacking Attacks

- Motivation and background
- Stack based buffer overflow attack
- Code reuse attacks
- Countermeasures
- Other Types Boundary Error Vulnerabilities

# Boundary Errors and Control-Hijack Attacks

- Why this lecture?
  - Stack problems are very common failures
  - The methods used help to understand program execution at a very low level
  - Performing a buffer overflow attack should sensitize everybody to avoid program weaknesses which enable these attacks

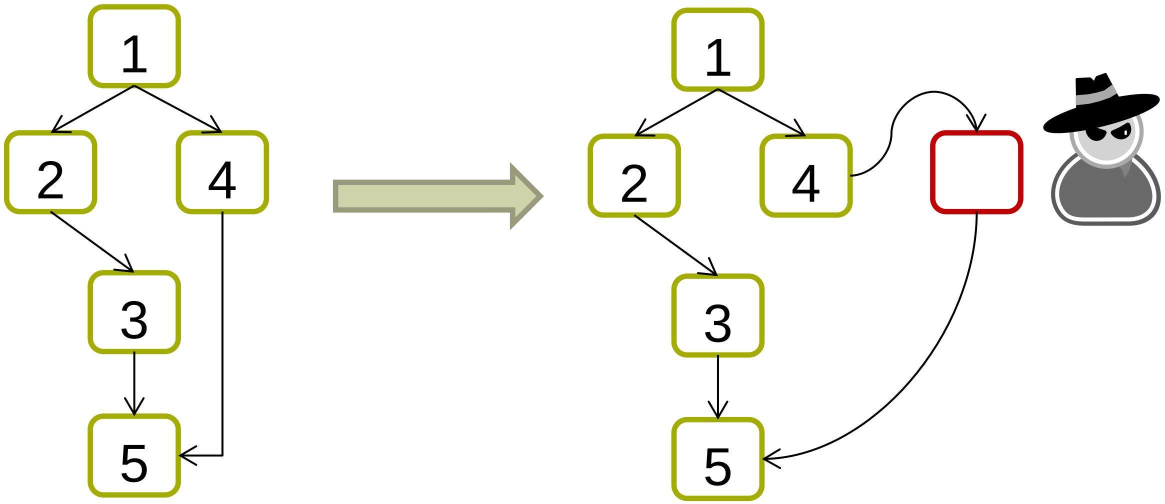
- What to learn?
  - How stack works in practice
  - How buffer overflow can be generated
  - How to reuse code for attacks

# Characteristics of Control-Hijack Attacks

- Runtime attacks in which an attacker is able to execute arbitrary (malicious) code in a device by manipulating the program control flow
- Subversion of the control flow is typically done by exploiting memory-related vulnerabilities
- These vulnerabilities are often found in programs written in memory-unsafe languages as C/C++ and Assembly

# Control-Hijack Exploitation

- A successful exploit involves:
  - Subverting the control flow of a program at runtime
  - Executing malicious code instead of the original intended code
    - Code Injection Attacks
    - Code Reuse Attacks




Black Hat image source: http://www.opensecurityarchitecture.org/cms/library/icon-library

# Severity of Control-Hijack Attacks

- Vulnerable code is the result of incorrect boundary checks
- Publicly known since the 70's
- Still a major threat in today's systems
  - Buffer Overflow vulnerabilities ranked as the overall most important vulnerability from 1988-2012 [1]
  - Classic Buffer Overflow errors ranked #3 at the 2011 CWE/SANS top 25 most dangerous software errors list [2]
  - In the 2020 CWE Top 25 "Out-of-Bounds Write" is ranked #2

[1] "25 Years of Vulnerabilities", accessed Sept 24, 2013, http://labs.snort.org/blogfiles/Sourcefire-25-Years-of-Vulnerabilities-Research-Report.pdf
[2] "2011 CWE/SANS Top 25 Most Dangerous Software Errors", accessed Sept 24, 2013, http://cwe.mitre.org/top25/

# CWE: Common Weaknesses Enumeration System

- Top 25: http://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

| Rank | ID | Name | Score |
|---|---|---|---|
| [1] | CWE-79 | Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') | 46.82 |
| [2] | CWE-787 | Out-of-bounds Write | 46.17 |
| [3] | CWE-20 | Improper Input Validation | 33.47 |
| [4] | CWE-125 | Out-of-bounds Read | 26.50 |
| [5] | CWE-119 | Improper Restriction of Operations within the Bounds of a Memory Buffer | 23.73 |
| [6] | CWE-89 | Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') | 20.69 |
| [7] | CWE-200 | Exposure of Sensitive Information to an Unauthorized Actor | 19.16 |
| [8] | CWE-416 | Use After Free | 18.87 |
| [9] | CWE-352 | Cross-Site Request Forgery (CSRF) | 17.29 |
| [10] | CWE-78 | Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') | 16.44 |
| [11] | CWE-190 | Integer Overflow or Wraparound | 15.81 |

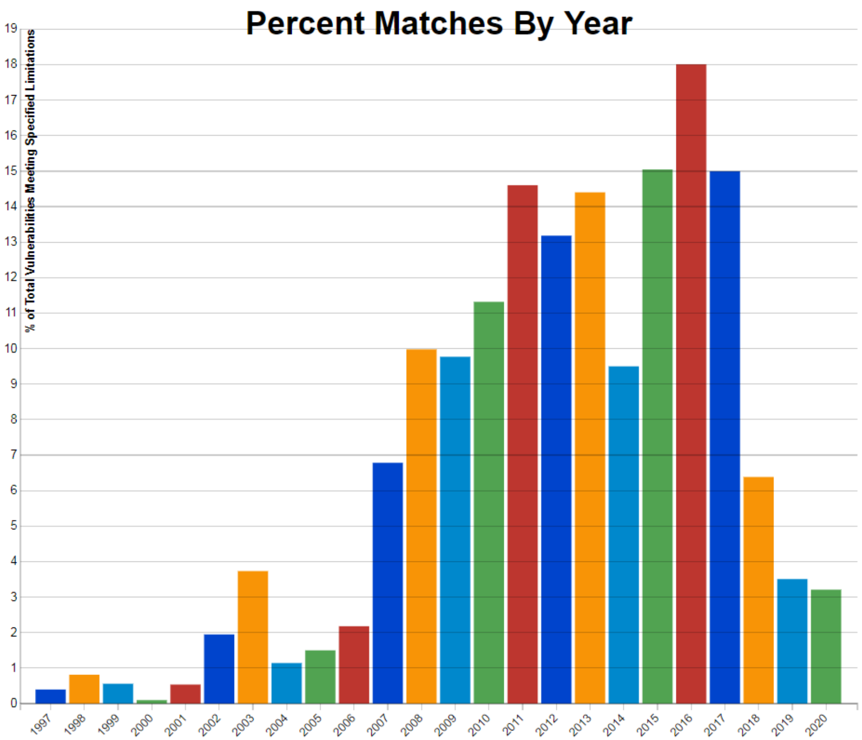
# CWE: Common Weaknesses Enumeration System

- Top 25: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html#tableView

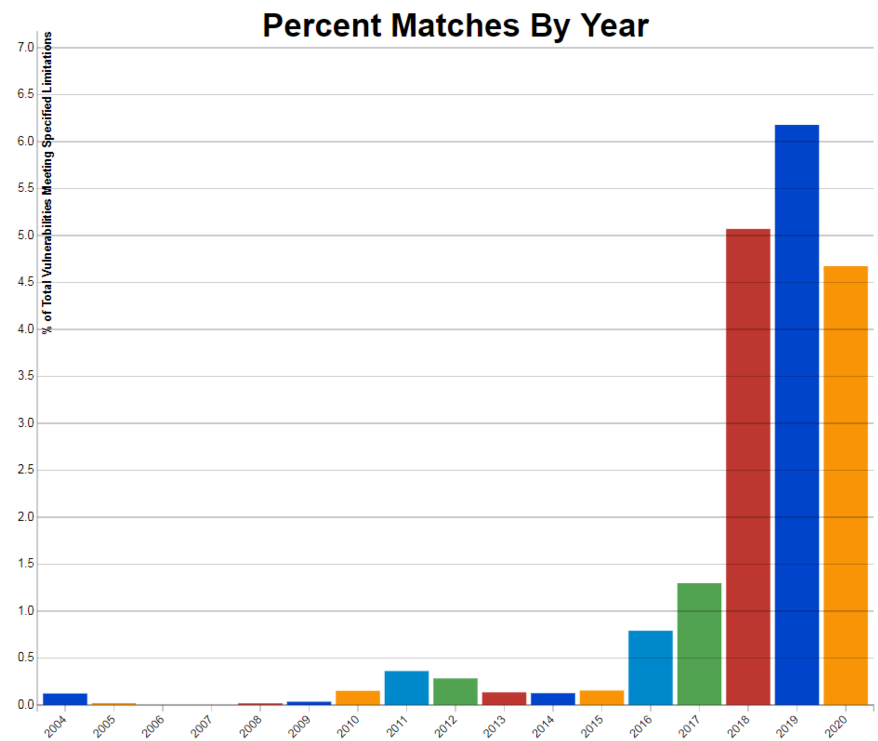
| Rank | ID | Name | Score | CVEs in KEV | Rank Change vs. 2022 |
|---|---|---|---|---|---|
| 1 | CWE-787 | Out-of-bounds Write | 63.72 | 70 | 0 |
| 2 | CWE-79 | Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') | 45.54 | 4 | 0 |
| 3 | CWE-89 | Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') | 34.27 | 6 | 0 |
| 4 | CWE-416 | Use After Free | 16.71 | 44 | +3 |
| 5 | CWE-78 | Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') | 15.65 | 23 | +1 |
| 6 | CWE-20 | Improper Input Validation | 15.50 | 35 | -2 |
| 7 | CWE-125 | Out-of-bounds Read | 14.60 | 2 | -2 |
| 8 | CWE-22 | Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') | 14.11 | 16 | 0 |
| 9 | CWE-352 | Cross-Site Request Forgery (CSRF) | 11.73 | 0 | 0 |
| 10 | CWE-434 | Unrestricted Upload of File with Dangerous Type | 10.41 | 5 | 0 |

# Vulnerabilities statistics related to buffer overflow 2020

CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer




A Child of CWE-119 is
CWE-787: Out-of-bounds Write

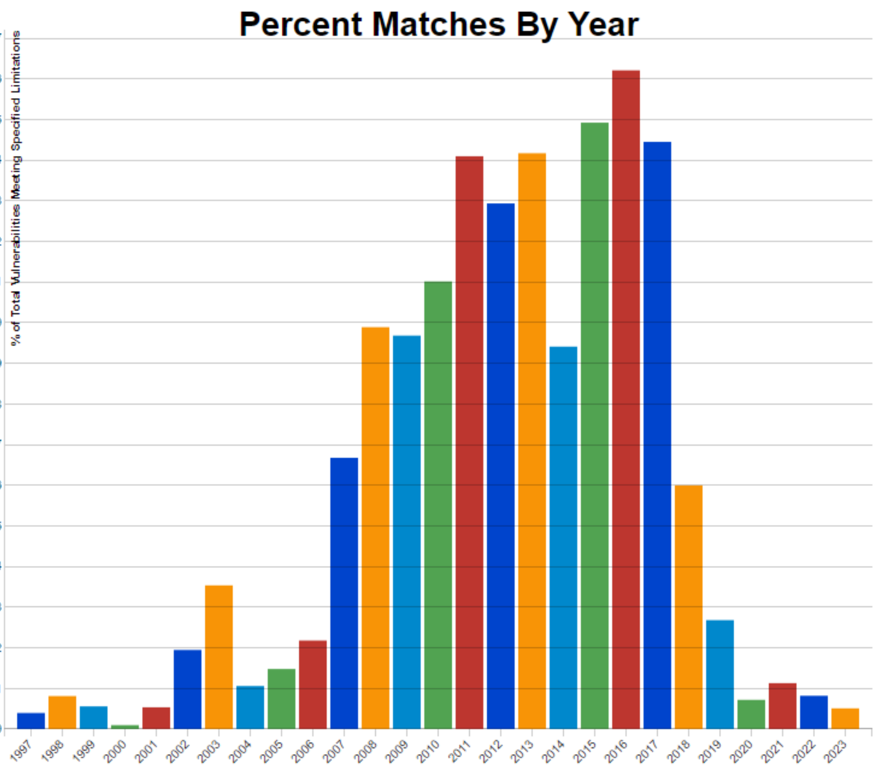



Source: National Vulnerability Database”, accessed Nov 20, 2020, http://nvd.nist.gov/ with queries:

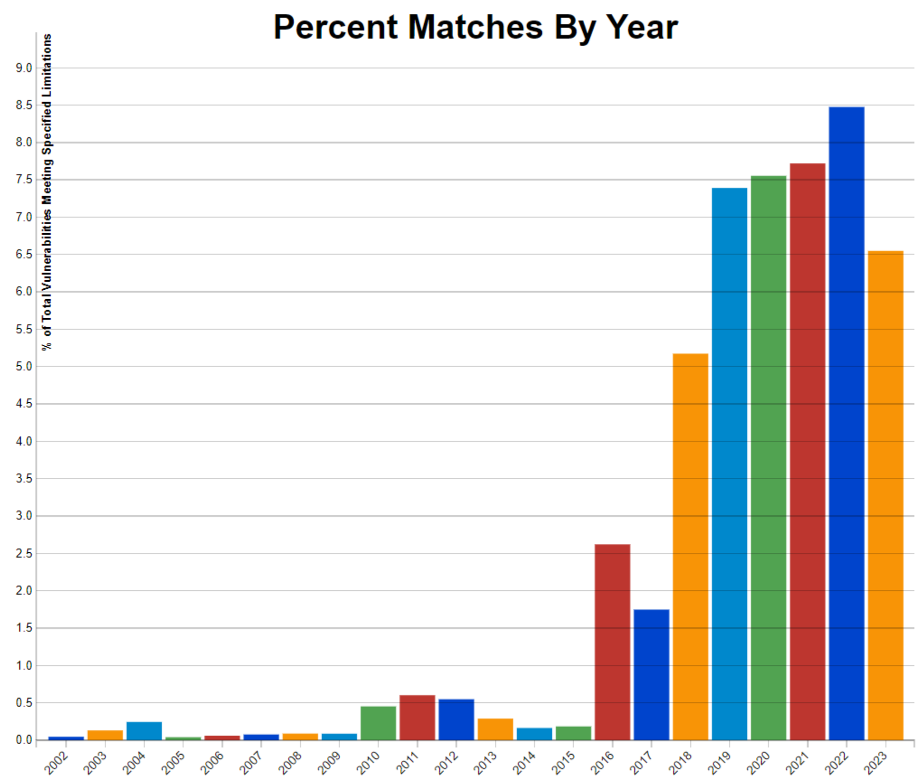
https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&search_type=all&cwe_id=CWE-119
https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&search_type=all&cwe_id=CWE-787

# Vulnerabilities statistics related to buffer overflow 2023

CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer




A Child of CWE-119 is
CWE-787: Out-of-bounds Write




Source: National Vulnerability Database", accessed Nov 11, 2023, http://nvd.nist.gov/ with queries:

https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&search_type=all&cwe_id=CWE-119
https://nvd.nist.gov/vuln/search/statistics?form_type=Advanced&results_type=statistics&search_type=all&cwe_id=CWE-787

# Buffer Overflow Attacks Timeline

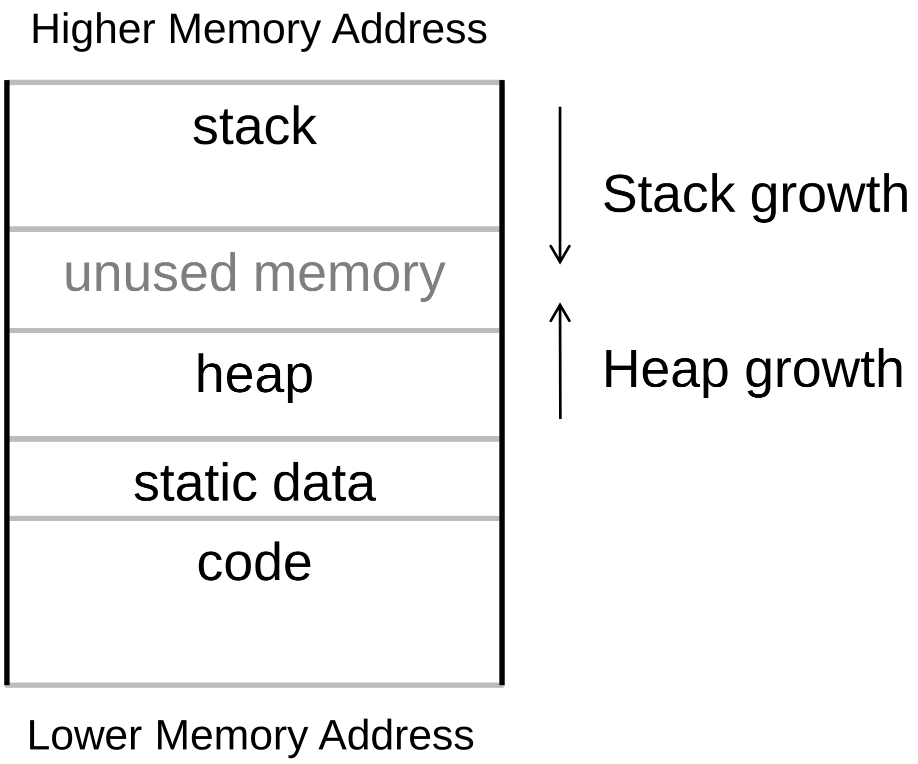
| | |
|---|---|
| 1972 | • First public documentation about buffer overflows |
| 1988 | • First documented hostile exploitation by the Morris worm |
| 1995 | • Initial rediscovery of buffer overflows on Bugtraq |
| 1996 | • Aleph One's Phrack paper Smashing the Stack for Fun and Profit |
| 1997 | • First return-into-library exploit by Solar Designer |
| 2000 | • Nergal's Phrack paper about advanced return-into-library exploits |
| 2001 | • First major worm that used buffer overflows (CodeRed) |
| 2005 | • Borrowed code chunks technique introduced by Sebastian Krahmer |
| 2007 | • Return-oriented programming introduced for the x86 by Hovav Shacham |
| 2008 | • Return-oriented programming for the first RISC architecture SPARC |
| 2009 | • First practical example of return-oriented programming |
| 2009 | • First example of return-oriented programming for the ARM architecture |

# Boundary Errors and Control Hijacking Attacks

- Motivation and background
- Stack based buffer overflow attack
- Code reuse attacks
- Countermeasures
- Other Types Boundary Error Vulnerabilities

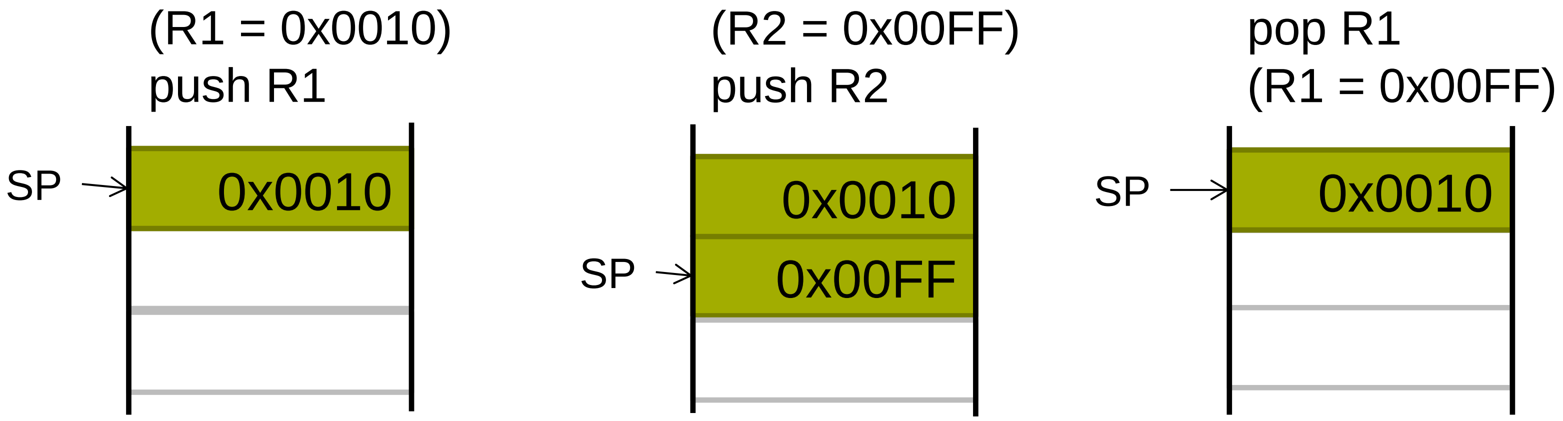
# Standard Memory Layout

- C compilers usually arrange the memory as follows:



# Review of the Stack

- Last-In First-Out data structure
- Items can be inserted into the stack with a PUSH
- Items can be retrieved from the stack with a POP
- A Stack Pointer (SP) is used to point to the element at the “top” of the stack

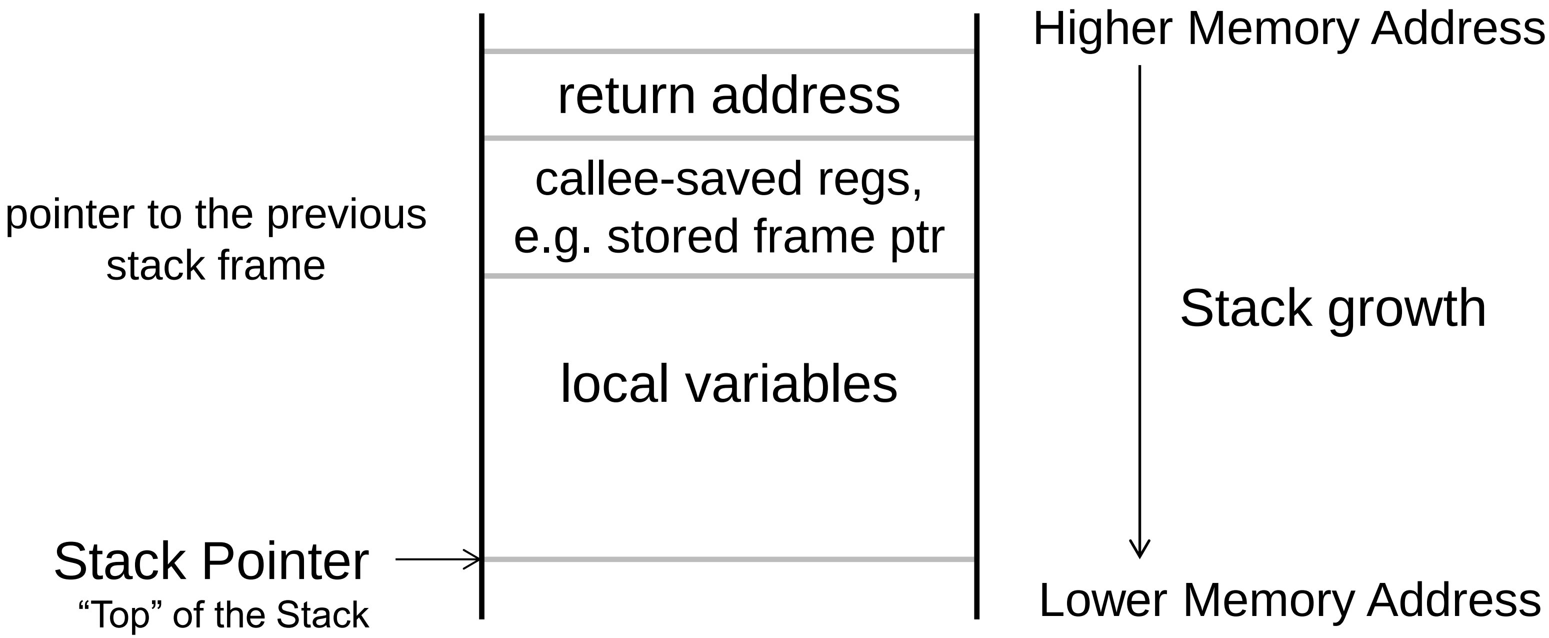


# Stack Frame

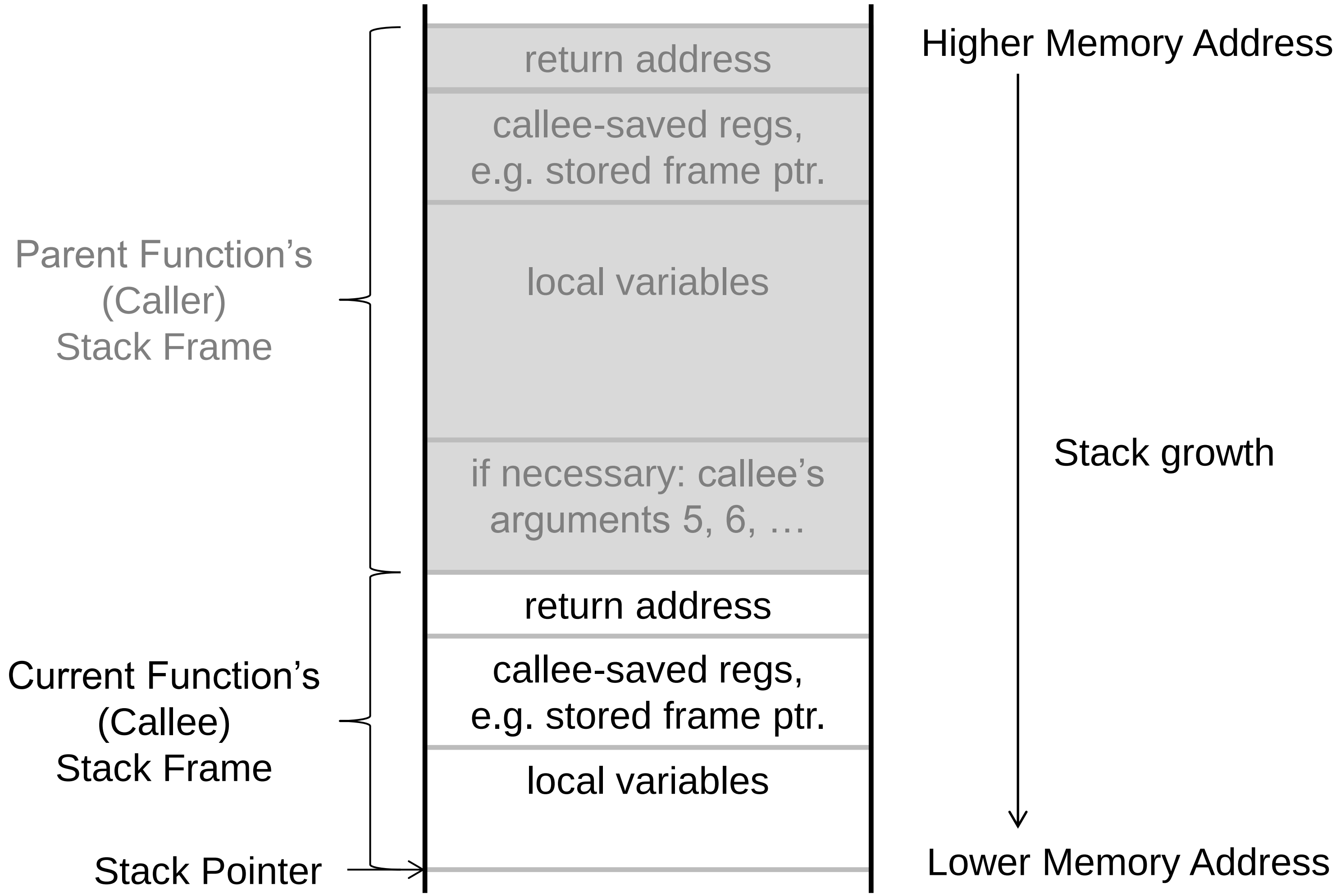
- A stack frame is a logical partition of the stack
- Every time a function is called a new stack frame is created
- The stack frame is used to
  - Allow passing variables between functions
  - Store variables from previous functions
  - Allow each subroutine to use the stack without disturbing other functions
  - Allow a subroutine to use the memory from the stack frame up to the end of the stack
- Entry and exit sequences (aka. prologue and epilogue) take care of managing the stack frames. The actual sequences depend on the compiler's calling conventions.

# Stack Frame

- A typical (function calls other functions with few (<=4) arguments) Stack Frame looks like this:



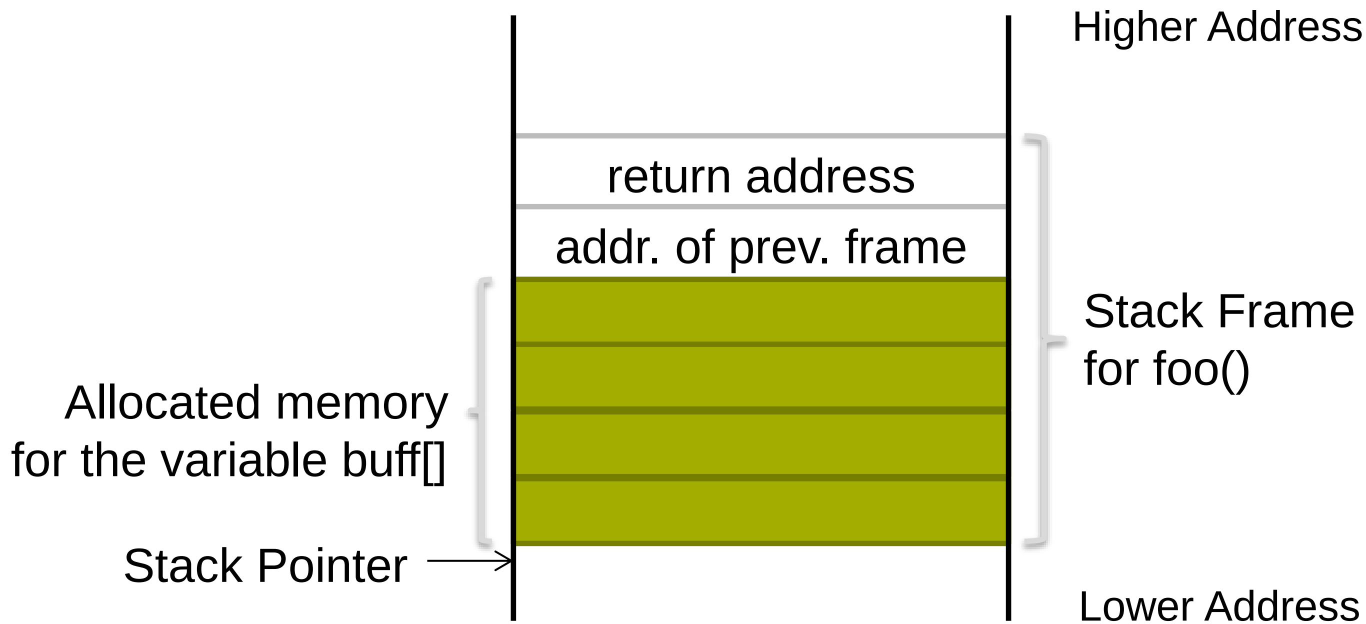
# Stack Frame


return address
callee-saved regs,
e.g. stored frame ptr.
local variables
if necessary: callee's
arguments 5, 6, …
return address
callee-saved regs,
e.g. stored frame ptr.
local variables
Parent Function's
(Caller)
Stack Frame
Current Function's
(Callee)
Stack Frame
Stack Pointer
Higher Memory Address
Stack growth
Lower Memory Address

# Stack-based Buffer Overflow

- If ranges are not checked, data may be written beyond the boundary of the memory allocated for a variable.
- This will overwrite any data stored in the stack.

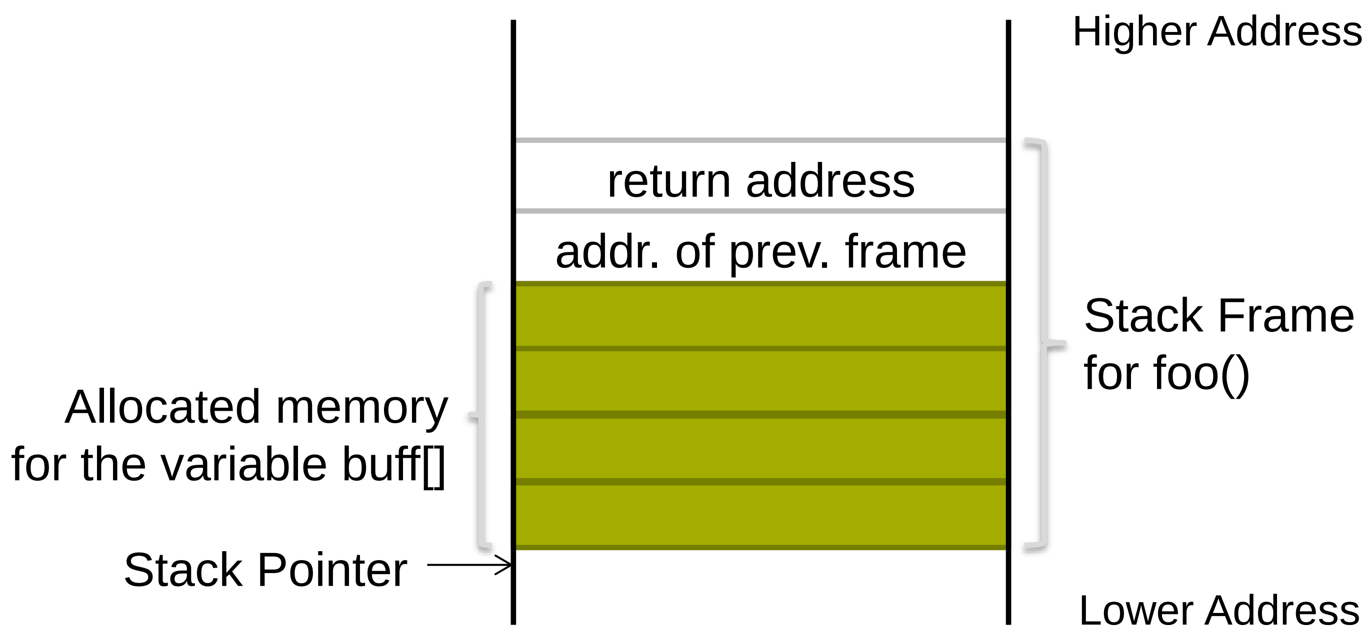
```
void foo (char *str)
{
   char buff[16];
   strcpy(buff, str);
}
```



# Stack-based Buffer Overflow

- The function *strcpy(dst, src)* will not check the boundaries of the buffer, it will
  - Copy each byte from the src address to dst address
  - Incrementing the pointers after each copy

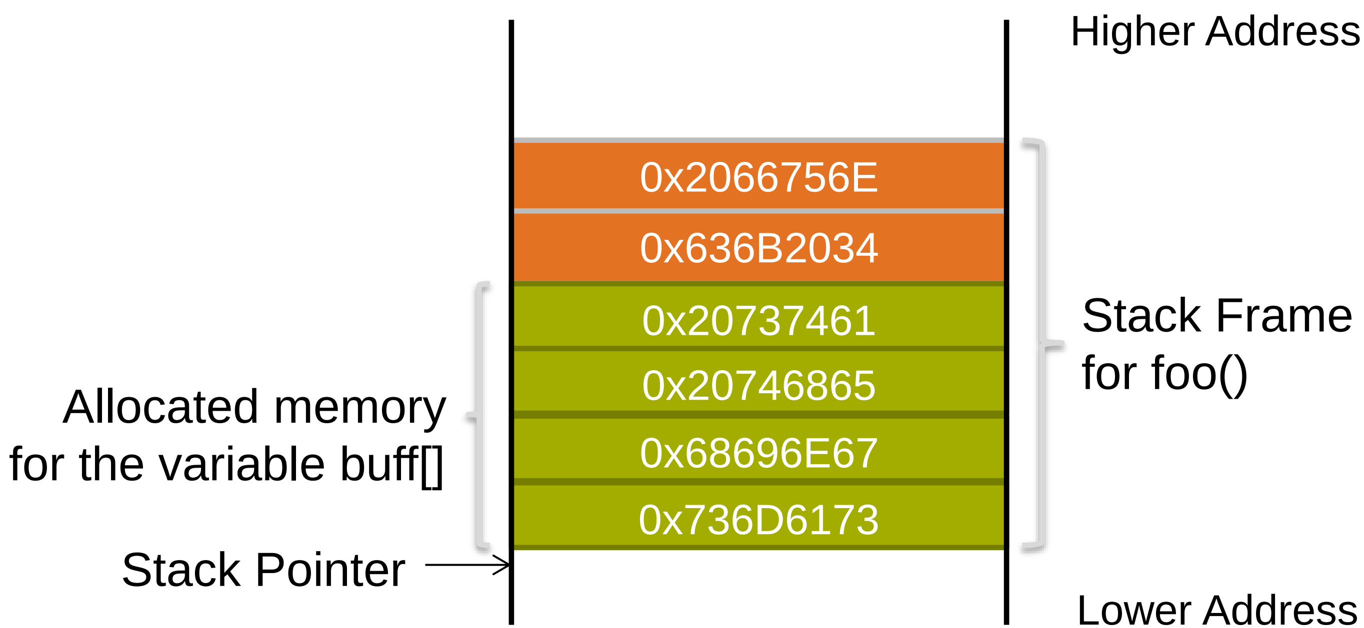
```
void foo (char *str)
{
   char buff[16];
   strcpy(buff, str);
}
```



# Stack-based Buffer Overflow

- Assuming that str = “smashing the stack 4 fun”
- The buffer boundary will be exceeded
- And frame pointer and return address will be overwritten!

```
void foo (char *str)
{
    char buff[16];
    strcpy(buff, str);
}
```



# Stack-based Buffer Overflow example

- Example in ARM using the *strcpy* function
- Shows the stack before copying the value of *str* into *buff*

```
void foo(char *str)
{
    char buff[8];
    strcpy(buff, str);
}

int main(void)
{
    foo("it will overflow");
    return 0;
}
```

Stack:

| address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 200007E0 | 00 | 00 | 00 | 00 | 88 | 12 | 00 | 08 |
| 200007E8 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 200007F0 | F8 | 07 | 00 | 20 | 09 | 10 | 00 | 08 |
| 200007F8 | 00 | 00 | 00 | 00 | 31 | 10 | 00 | 08 |

space allocated for *buff*

return address in main

frame pointer

# Stack-based Buffer Overflow example

- Example using the *strcpy* function
- Shows the stack **after** copying the value of *str* into *buff*

```
void foo(char *str)
{
    char buff[8];
    strcpy(buff, str);
}

int main(void)
{
    foo("it will overflow");
    return 0;
}
```

Stack:

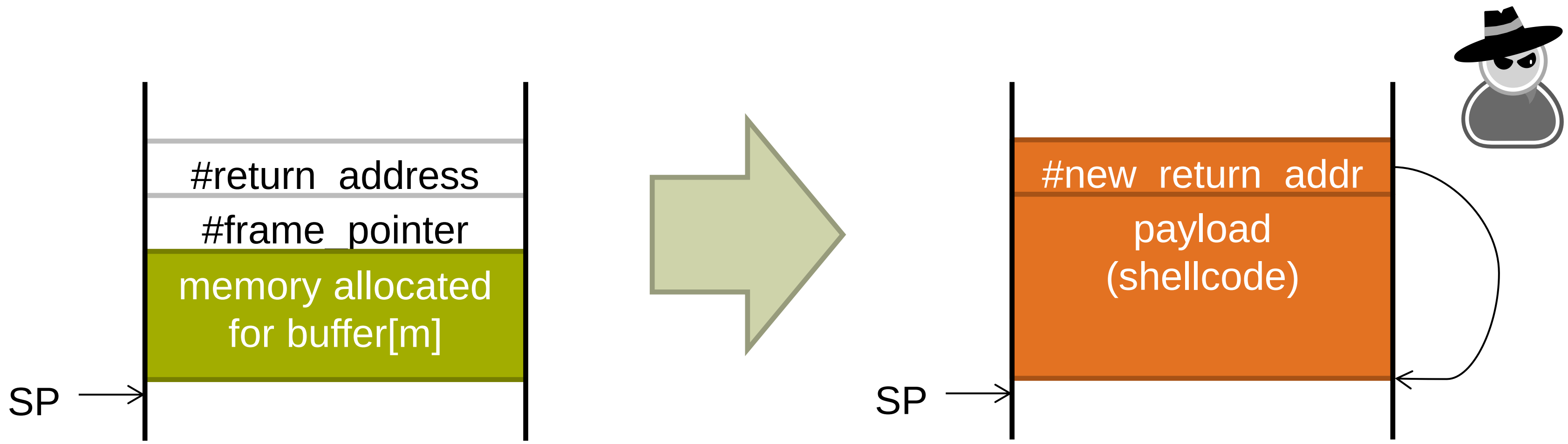
space allocated for *buff*

| address | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 200007E0 | 00 | 00 | 00 | 00 | 88 | 12 | 00 | 08 |
| 200007E8 | 69 | 74 | 20 | 77 | 69 | 6C | 6C | 20 |
| 200007F0 | 6F | 76 | 65 | 72 | 66 | 6C | 6F | 77 |
| 200007F8 | 00 | 00 | 00 | 00 | 31 | 10 | 00 | 08 |

frame pointer    return address in main

# Stack-based Buffer Overflow Code Injection

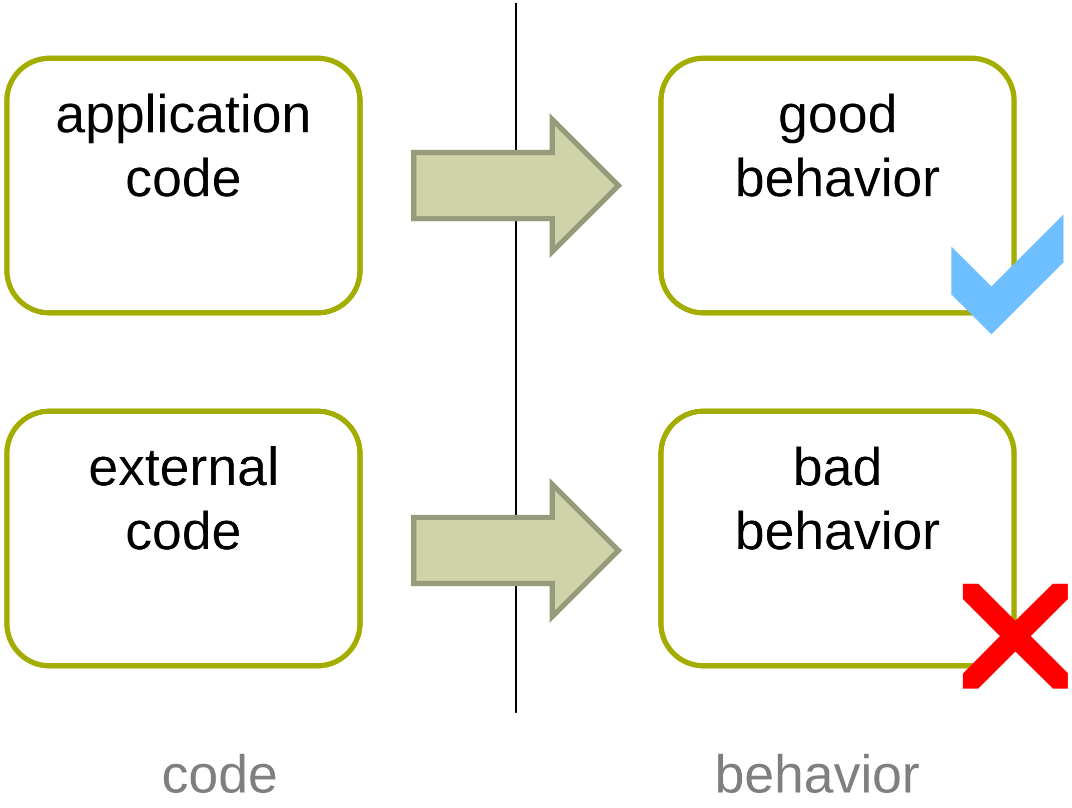
- An attacker may use buffer overflows to inject his code into the stack:
  - A payload is written in machine code and injected to the stack
  - The return address is overwritten to point to the injected code
- Historically this type of programs receive the name of shellcodes



# Boundary Errors and Control Hijacking Attacks

- Motivation and background
- Stack based buffer overflow attack
- Code reuse attacks
- Countermeasures
- Other Types Boundary Error Vulnerabilities

# Code Behavior Assumptions



# Code Reuse Attack

- General concept
  - Execute malicious software without code injection
  - Make use of existing libraries or instruction sequences

- Advantages (for an attacker)
  - No code needs to be injected
  - Trusted code can be used for malicious purposes

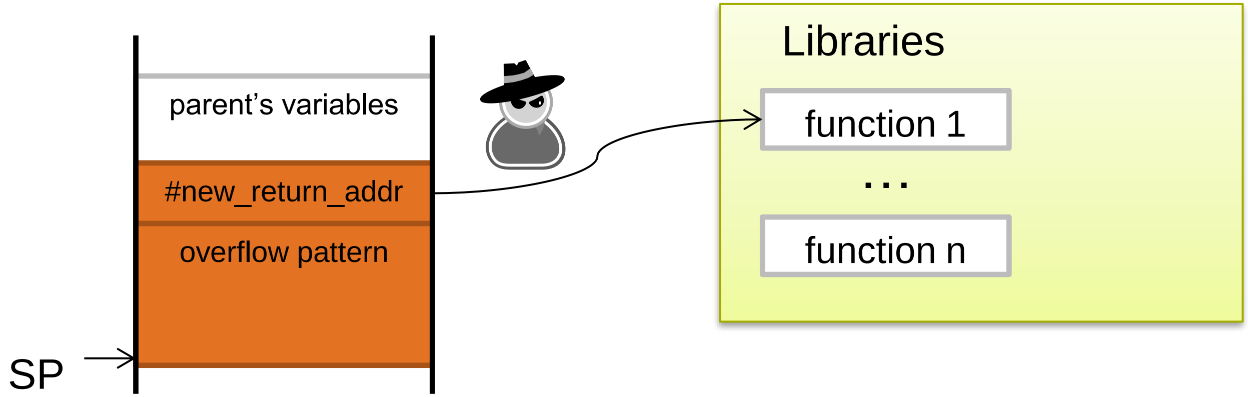
- Limitations
  - An attacker relies on available functions or instruction sequences

# Types of Code Reuse Attacks

- Return-into-libc
  - uses complete functions present in the code being attacked
- Return-into-libc chaining with ret/pop
  - chains several functions to perform more complex behavior
- Borrowed code chunks
  - uses short assembly code sequences ending in ret
  - chains them together by controlling the return address
- Return-Oriented Programming
  - uses chained code snippets called gadgets to perform a specific function
  - demonstrated Turing completeness without code injection

# Return-into-libc

- Exploit concept
  - An attacker changes the return address in stack to point to a function which already exists (libc)
  - By modifying the parameters sent to the functions an attacker may change the functionality of the program
  - To form more complex attacks, functions can be linked to be executed in a chain



# Return-Oriented Programming

- Idea
  - Induce arbitrary behavior in a target system without code injection
  - By locating short code snippets already present in the code and linking them in a malicious order

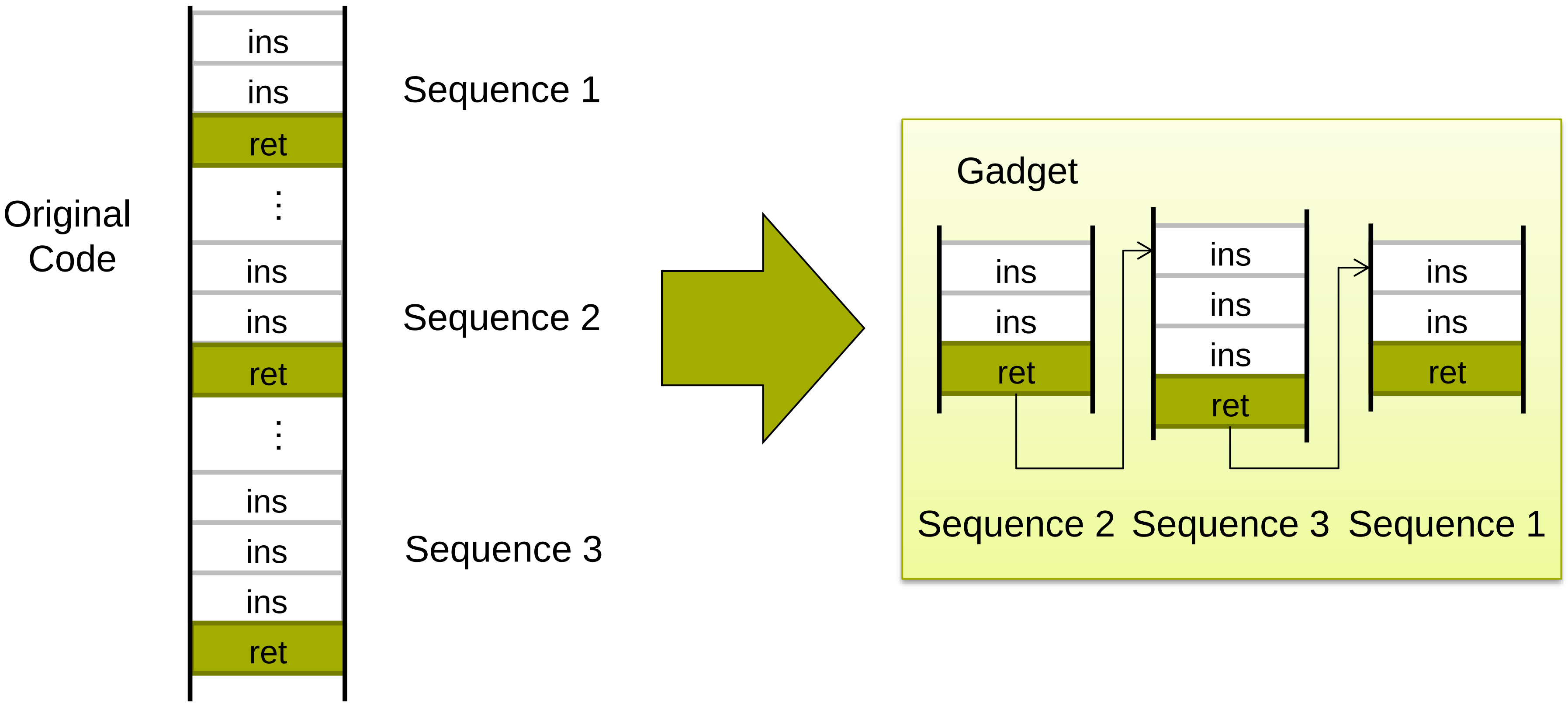
- Contributions
  - May be used to define a Turing-complete language
  - Likely possible in every architecture
    - Known for x86, SPARC, Atmel AVR, Z80, Power PC, and ARM

# Return-Oriented Programming

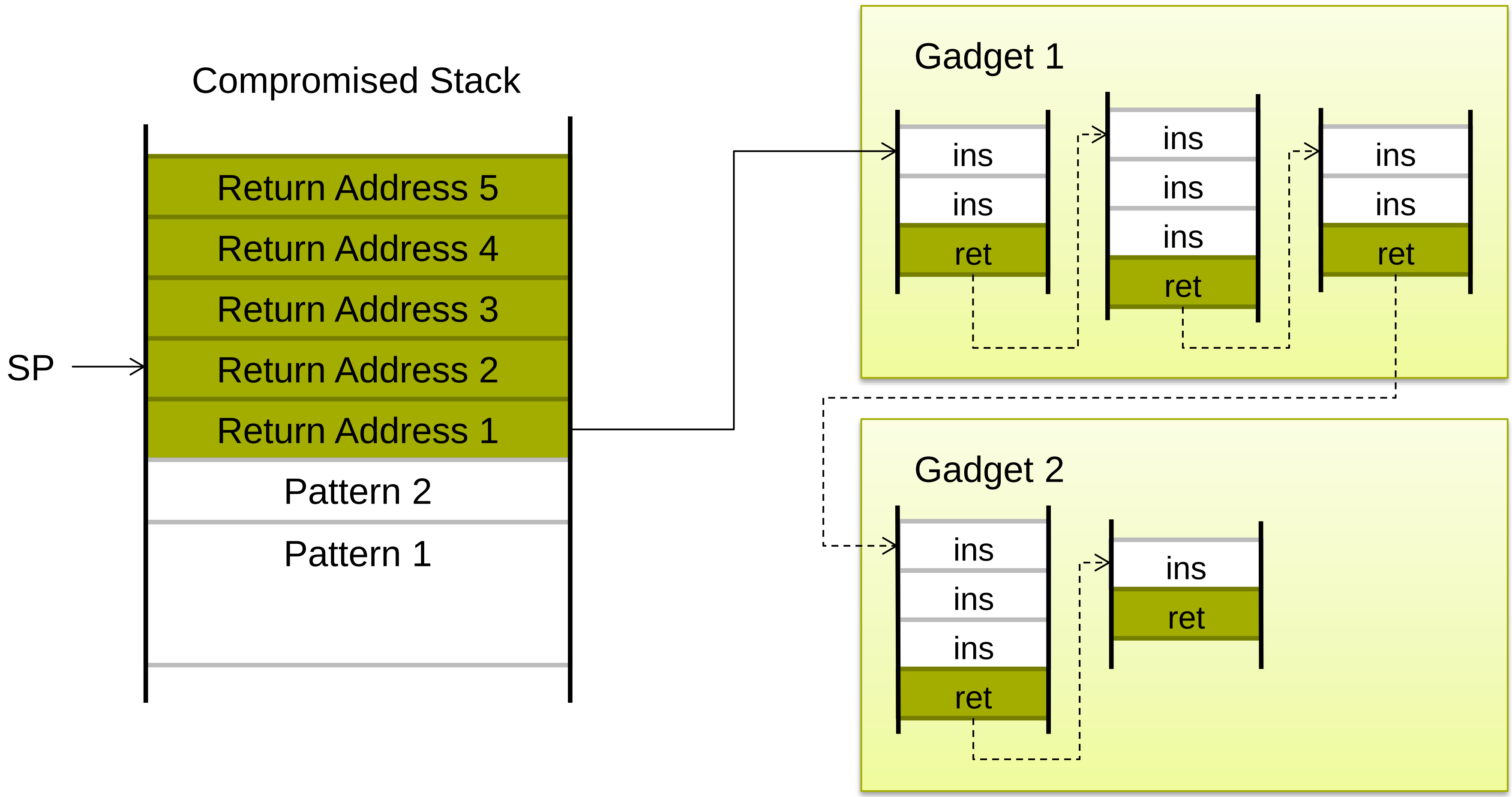
- Turing-complete language based on gadgets

- Gadgets
  - Small instruction sequence ending with a RET instruction
  - Sequences are chained together with the RET instruction to form a “gadget”
  - Each gadget performs a specific computation task (load, store, arithmetic operations, etc.)
  - The combination of gadgets creates arbitrary programs which the attacker can execute

# Gadgets


Original Code
ins
ins
ret
⋮
ins
ins
ret
⋮
ins
ins
ins
ret
Sequence 1
Sequence 2
Sequence 3
Gadget
ins
ins
ret
ins
ins
ins
ret
ins
ins
ret
Sequence 2
Sequence 3
Sequence 1

# ROP Attack Example



# Gadget Elements

- A gadget is composed of the following elements
  - Gadget operation
    - Arithmetic/Logic, Load/Store, Branches, etc…
  - Chaining variables
    - Registers or memory locations which can be controlled (e.g. loading a register with data injected in the stack)
  - Side-effect variables
    - Variables which are modified unintentionally
    - May require the use of another gadget in order to “clean” them
  - Chaining instruction
    - Return instruction
      - usually RET
      - can be any RET-like instruction (e.g. pop PC)

# Gadget Elements

- Example for the ARM architecture
  - Register Shift-Left operation

Assembly Code:

```
LSL     R2, R2, R12
SUB     R1 , R1 , R12
ADD     R1 , R1 , 1
LDMFD   SP!, {R3 ,PC}
```

Gadget Elements:

| | |
|---|---|
| Gadget Operation | R2 = R2 << R12 |
| Chaining Variables | R3 = MEM [SP + 0] |
| Gadget Chaining | PC = MEM [SP + 4] |
| Side-effects | R1 |

# Gadget Elements

- Example: R2 = 1 << 4;

Assembly Code:

```
...
0x0310  LSL     R2, R2, R12
0x0314  SUB     R1 , R1 , R12
0x0318  ADD     R1 , R1 , 1
0x031C  LDMFD   SP!, {R3 ,PC}
...
0x201C  LDMFD   SP!, {R2 ,PC}
...
0x5200  LDMFD   SP!, {R12 ,PC}
```

Compromised Stack

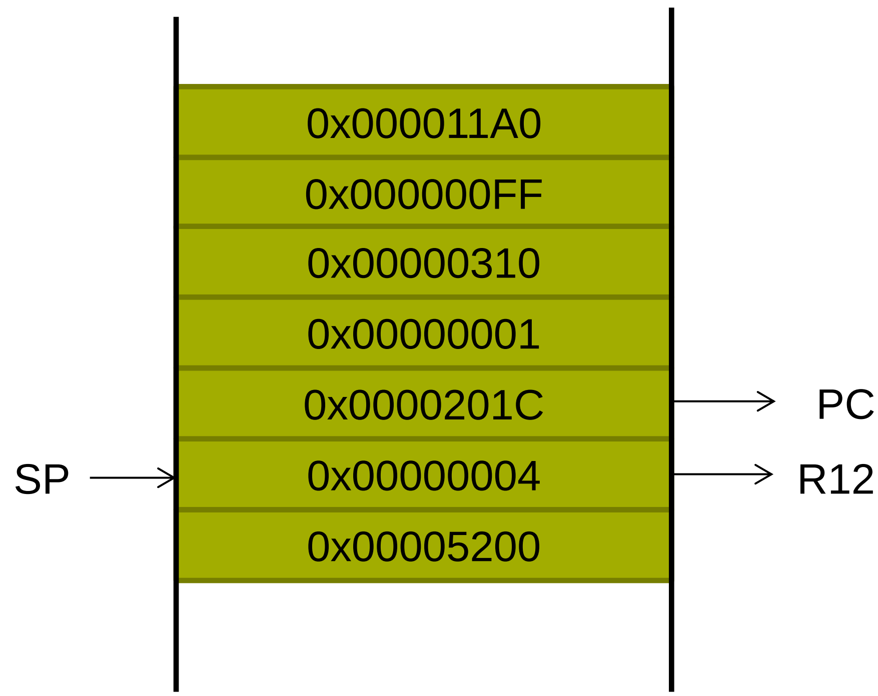
# Gadget Elements

- Example: R2 = 1 << 4;

Assembly Code:

```
...
0x0310  LSL     R2, R2, R12
0x0314  SUB     R1 , R1 , R12
0x0318  ADD     R1 , R1 , 1
0x031C  LDMFD   SP!, {R3 ,PC}
...
0x201C  LDMFD   SP!, {R2 ,PC}
...
0x5200  LDMFD   SP!, {R12 ,PC}
```

Compromised Stack



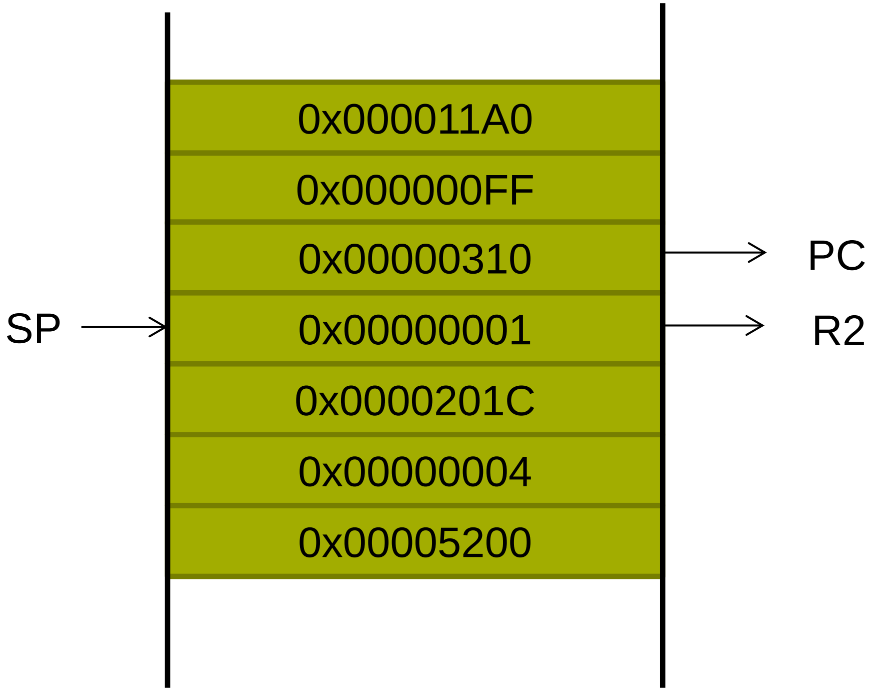
# Gadget Elements

- Example: R2 = 1 << 4;

Assembly Code:

```
...
0x0310  LSL     R2, R2, R12
0x0314  SUB     R1 , R1 , R12
0x0318  ADD     R1 , R1 , 1
0x031C  LDMFD   SP!, {R3 ,PC}
...
0x201C  LDMFD   SP!, {R2 ,PC}
...
0x5200  LDMFD   SP!, {R12 ,PC}
```

Compromised Stack



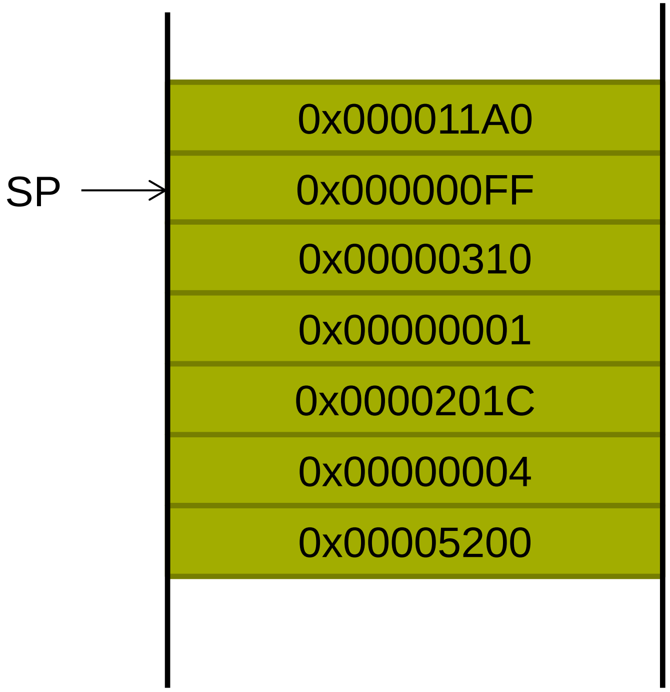
# Gadget Elements

- Example: R2 = 1 << 4;

Assembly Code:

```
...
0x0310  LSL     R2, R2, R12
0x0314  SUB     R1 , R1 , R12
0x0318  ADD     R1 , R1 , 1
0x031C  LDMFD   SP!, {R3 ,PC}
...
0x201C  LDMFD   SP!, {R2 ,PC}
...
0x5200  LDMFD   SP!, {R12 ,PC}
```

Compromised Stack



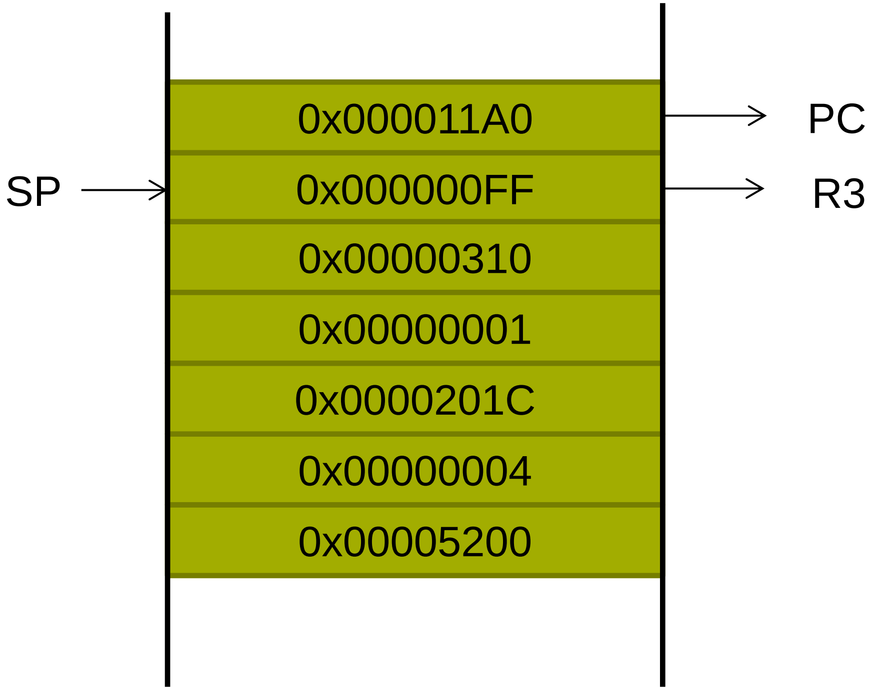
# Gadget Elements

- Example: R2 = 1 << 4;

Assembly Code:

```
...
0x0310  LSL     R2, R2, R12
0x0314  SUB     R1 , R1 , R12
0x0318  ADD     R1 , R1 , 1
0x031C  LDMFD   SP!, {R3 ,PC}
...
0x201C  LDMFD   SP!, {R2 ,PC}
...
0x5200  LDMFD   SP!, {R12 ,PC}
```

Compromised Stack



# Generating Return-Oriented Programs

- Methodology
  - Scan common libraries for useful instruction sequences ending in gadget chaining instructions (e.g. ret)
  - Chain instruction sequences in order to form the desired gadgets (e.g. a Turing-complete catalog)
  - Create a payload list of the addresses of the gadgets and any values used for computations
  - Introduce the payload into the stack
  - Point the Stack Pointer into the first address of the payload

# Boundary Errors and Control Hijacking Attacks

- Motivation and background
- Stack based buffer overflow attack
- Code reuse attacks
- Countermeasures
- Other Types Boundary Error Vulnerabilities

# Mitigation

- NX segments
- Canaries
- Shadow stacks
- Control Flow Integrity
- Address Space Layout Randomization

# NX Segments

- Main concept
  - Non eXecutable (NX) segments prevent the execution of (injected) code from data segments (e.g. stack)
  - Segments may be marked as either writable or executable but never as both (W^X)
  - Provides a strict separation between code and data
- Drawbacks
  - Does not prevent buffer overflows
  - An attacker can no longer execute injected code … but can still corrupt the stack
  - Code reuse attacks such as ROP and Return-into-libc can still be performed

# Canaries

- Main concept
  - Inserting hard-to-predict patterns, called canaries, to guard the control-flow data (e.g. return addresses) in the stack
  - During a buffer overflow the canaries will be overwritten with a different value
  - The program is terminated if the canary in the stack is corrupted and does not match an expected value.
- Drawbacks
  - Indirect pointer overwrites[4] may corrupt the return address while maintaining the integrity of the canary

[4] Bulba and Kil3r. Bypassing Stackguard and Stackshield. Phrack , 56, 2000.

# Shadow Stacks

- Main concept
  - When a function is entered the return address is copied to a different location called the shadow stack
  - The value of the function’s return address in the stack is checked against the copy in the shadow stack
  - In the event of a discrepancy the program is terminated
- Drawbacks
  - Local variables and function arguments are not protected
  - Vulnerable to non-control data attacks (i.e. buffer overflows which target data)

# Control Flow Integrity (CFI)

- Main concept
  - Create a control flow graph (CFG) for the program
  - Label all nodes with a unique ID
  - Insert instructions which at runtime check if the branch into a node was valid by comparing against the expected ID
- Drawbacks
  - Comparing node IDs presents an overhead that affects code size and performance
  - Cannot stop attacks focused on corrupting data (data-only attacks)

# Address Space Layout Randomization (ASLR)

- Main concept
  - Randomize the base address of each segment (stack, heap, code, etc…)
  - The attacker does not know the exact location of the instruction sequences
- Drawbacks
  - If parts of the code cannot be randomized because of the device architecture, the attacker may still use this information to construct gadgets.
  - Brute force attacks are possible

# Boundary Errors and Control Hijacking Attacks

- Motivation and background
- Stack based buffer overflow attack
- Code reuse attacks
- Countermeasures
- Other Types Boundary Error Vulnerabilities

# Other Types Boundary Error Vulnerabilities

- Even though stack-based buffer overflows are the most common type of vulnerabilities caused by boundary errors, they are by no means the only type

- Other common types of buffer overflows include
  – Heap-based buffer overflows
  – Format string attacks
  – Integer overflows

# Integer Overflows

- Variables store values which should not exceed the largest value that they can contain
- Exceeding the largest value will cause the variable to wrap around
- The variable will wrap around to a small value or a negative number
- If the wrap is unexpected it can lead to a security vulnerability
- Critical if the result is used to:
  - Control a loop
  - Make security decisions
  - Determine an offset or size (e.g. during memory allocation)
  - Copy data

# Format String Vulnerabilities

- This type of vulnerabilities occurs when using format functions that do not control the string format
- Format functions are functions which allow a string to be formatted to a human readable representation
- Typical ANSI C format functions include the *printf() family of functions
- If an attacker is able to provide the string format to the format function a vulnerability may be exploited

# Format String Vulnerabilities

- When is a format function vulnerable?

**Vulnerable code**

…
printf(data);
…

The string format is controlled by the content of the data!

**Correct code**

…
printf("%s", data);
…

The string format is controlled by the function

# Format String Vulnerabilities

- What type of attacks can be conducted?

  - Read out the stack
    - “%s” reads a string
    - “%x” reads a hex value
  - Write to the stack
    - “%n” writes the number of printed characters to a pointer to an integer (which an attacker can control)
  - Further attacks by controlling the stack
    - Read out any memory location
    - Overwriting arbitrary memory
    - Control Flow Hijacking

# Conclusions

- Even after over 25 years of knowing about boundary error exploitations, they are still one of the most dangerous software errors

- Stack-corruption vulnerabilities are still one of the major threats to the security of a device, but the exploitation of other types of vulnerabilities is increasing as techniques to bypass known protection mechanisms are continuously being developed

- The paradigm of protecting a system has changed from preventing the injection of malicious *code* to preventing malicious *actions* even when the code is assumed to be benign

# References


- “VUDO Malloc Tricks”, MaXX, Phrack Magazine, issue 57, August 2001
- “Once Upon a Free”, Anonymous, Phrack Magazine, issue 57, August 2001
- “Smashing the Stack for Fun and Profit”, Aleph One, Phrack Magazine, issue 49, August 1996
- “The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86)”, Hovav Shacham (ACM CCS 2007)
- “Return Oriented Programming for the ARM. Architecture”. Tim Kornau. December 22, 2009. Diplomarbeit. Ruhr-Universität Bochum.

# References


- “Exploiting Format String Vulnerabilities”, scut / team teso, September 1, 2001
- “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique”, S. Krahmer, September 2005
- “Address Space Layout Randomization”, PaX Team, http://pax.grsecurity.net/docs/aslr.txt, March 2003
- “Basic Integer Overflows”, Blexim, Phrack Magazine, issue 60, December 2002
- “Control-flow integrity,” M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, ACM Conference on Computer and Communications Security (CCS), 2005