Hacker&co.

or

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

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...

```c
#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...

```c
#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...

```c
#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...

```c
#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...

```c
#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 simple buffer overflow

HEX= 0x0000002a
STRING = "\x2a\x00\x00\x00"

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

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 it calls the attacker, who acts as a server

several available on the web

- [http://shell-storm.org/shellcode/](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

```
\x31\xc0\xb0\x19\x50\xcd\x80\x50 \\
\x50\x31\xc0\xb0\x19\x50\xcd\x80
```

// setreuid(geteuid(),getuid());

```
\xeb\x0d\x5f\x31\xc0\x50\x89\xe2 \\
\x52\x57\x54\xb0\x3b\xcd\x80\xe8
```

// exec(/bin/sh)

```
\xe6\xff\xff\xff/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 \(\rightarrow\) segmentation fault

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

check all with: objectdump -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 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

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 loosing 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)

```plaintext
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

<table>
<thead>
<tr>
<th>gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr1:</td>
</tr>
<tr>
<td>addr2:</td>
</tr>
<tr>
<td>addr3:</td>
</tr>
<tr>
<td>addr4:</td>
</tr>
</tbody>
</table>

buffer overflow

addr1
addr2
addr3
addr4
...
BP
SP

ROP: an example

<table>
<thead>
<tr>
<th>gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr1:</td>
</tr>
<tr>
<td>addr2:</td>
</tr>
<tr>
<td>addr3:</td>
</tr>
<tr>
<td>addr4:</td>
</tr>
</tbody>
</table>

buffer overflow

addr1
addr2
addr3
addr4
...
BP
SP

buffer overflow
ROP: an example

gadgets

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

buffer overflow

addr1
addr2
addr3
addr4
...

SP

BP

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 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?