fixing bugs in binaries using r2

@noconname

pancake
pancake@nopcode.org

The problem

Starting from the fact that there's no perfect or secure software. Different approaches are taken to workaround those security vulnerabilities.

- virtual machines
- sandboxes and virtualization
- prevention (libsafe, valgrind, ..)
- system level preventions (ASLR, PAX, SELinux, ..)
- software updates or source patches

On privative software, the situation is even worst, because many times the patches or updates appears late.

The solution

When we have no access to the source; the only way to fix that flaw is by patching the binary in order to fix or workaround the vulnerability.

Some community patches that have appeared before the official ones:

- iOS PDF (CVE 2010-1797)
- W32 LNK (CVE 2010-2568)
- W32 SMB (CVE 2009-3103)

...

The tools

We are going to use radare2 in order to analyze binaries and create patches.

The website of the project is:

```
http://www.radare.org
```

To get it from mercurial:

```
$ hg clone http://radare.org/hg/radare2
$ cd radare2
$ ./configure --prefix=/usr
$ make
$ sudo make symstall
```

Introduction

When a 0day vulnerability is disclosed, attackers focus on them in order to exploit it and lately many of them have been found in some trojans, worms or jailbreaks.

The techniques explained in this presentation have been useful in the following situations:

- capture the flag (protecting server)
- secure your phone (against pdf exploits)
- protect the desktop (windows box + dll hijack)

Patches

There are several ways to patch a vulnerability of a program or library.

- patch the binary (program or library)
- in-memory patch (attaching with debugger)
- preload library to override problematic symbols
- triggers (hooking events to patch on runtime)

We must have in mind that our patches must not corrupt stack or register state in order to make the program happy.

Patches: r2 commands

Here there are some commands useful to modify the code in memory or disk, assembling opcodes, inserting files and so on:

```
[0x00000000]> /x 383d3d3d3d4   # find watermark
[0x00034000]> f code @ ‘du -b code.bin | cut -f 1‘
[0x00034000]> !rasm2 pusha
60
[0x00034000]> wa pusha
[0x00034001]> s+1
[0x00034001]> wf code.bin @ $$+1
[0x00034001]> wa popa @ $$+1+code
```

Patches: in place

Patching the binary is probably the more clean solution, while not requiring extra dependencies to run.

```
push ebp                    push ebp
mov esp, ebp                push esp, ebp
sub esp, 40          ->     sub esp, 31337
```

If the patch code is bigger than the original we will need to find a place to put our code and branch the first bytes of the function into our wrapper.

```
...                  ->     push 32
push [ebp+4]                push [ebp+4]
push [esp-4]                push [esp-4]
call strcpy          ->     call strncpy
```

Patches: in place (2)

Finding a nest for our binary toy is important when the patched code is larger than the original.

- Rewrite function (optimize code by space)
- Relocate function somewhere else
- Overwrite unused symbols
- Abuse function paddings

Patches: preloading

Identify the library symbol to patch

- Check code references (xrefs) to given symbol
- Wrap the function with protection checks
- Check caller address or parameters
- Run original function

Only works for dynamically linked binaries

- Alternatives for W32, OSX, BSD/GNU

```
$ hg clone http://hg.youterm.com/toys/
$ cd toys/libwipe ; make
$ LD_PRELOAD=$PWD/libwipe.so thunderbird
```

Patches: in memory

Patching the process memory when the program is running becomes another way to fix vulnerabilities.

Must be applied in every execution.

- Using a debugger
- Exploiting the vulnerability
- Using triggers (plugins, ..)

```
$ echo wx 909090 | r2 -w -d ‘pidof chromium‘
```

Patches: triggers

Runtime patches can be triggered in many ways:

- function call
- exploiting, patch, restore stack
- import hooking (preloading)
- reusing symbols
- plugins (if application have them)

Trigger example: redsn0w

One of the several exploits to unlock iOS-based devices use this technique to patch the bootloader code in runtime.

In DFU mode, the device loads an old bootloader which permits memory writing, then patches the signature check.

```
arm7_stop                          # stop 2nd CPU
mw 0x9000000 0xe59f3014            #  \
mw 0x9000004 0xe3a02a02            #  |
mw 0x9000008 0xe1c320b0            #  |- write shellcode at
mw 0x900000c 0xe3e02000            #  |   address 0x900000
mw 0x9000010 0xe2833c9d            #  |
mw 0x9000014 0xe58326c0            #  |
mw 0x9000018 0xeafffffe            #  |
mw 0x900001c 0x2200f300            #  /
arm7_go                            # enable arm7, execute code
!sleep 1                           # wait a bit
arm7_stop                          # stop arm7 cpu
```

Trigger example: redsn0w

Disassembly of the payload with rasm2:

```
$ rasm2 -o 0x9000000 -e -a arm -d e59f3014e3a02a \
  02e1c320b0e3e02000e2833c9de58326c0eaffffe2200f300

ldr r3, [pc, #20]      ; 0x0900001c
mov r2, #8192          ; 0x2000
strh r2, [r3]
mvn r2, #0 ; 0x0
add r3, r3, #40192     ; 0x9d00
str r2, [r3, #1728]
b 0x09000018           ; jmp $$
```

Exploits the fact that the bootloader can write in memory and control the 2nd CPU which shares the RAM.

Using rarc2 to create patches

C-like language compiler that is included in r2 >= 0.6

It generates relocatable assembly code for intel/at&t x86-32/64 and ARM (thumb/normal)

```
$ rarc2 -aarm          # generate arm assembly
$ rarc2 -s -ax64       # GAS-compatible x86-64 assembly
```

```
$ cat hi.r
printf@alias(0x8048400);
main@global(,32) {
        printf("Hello World\n");
}

$ rarc2 hi.r > hi.rasm

$ rasm2 -a x86.olly -f hi.rasm
558bec81ec40000000c785ecffffff48656c6cc785f0ffffff6f20576f
c785f4ffffff726c640ac785f8ffffff00000008d85ecffffff8985fc
ffffffffb5fcffffffe8b8c37bf881c40400000081c4400000005dc3
```

Meld up everything with rapatch

rapatch is a program that uses **libr** api in order to ease the process of maintaining patches for binaries.

Integrates many key features of r2 in 90 LOC:

- rarc2: high-level language for patching
- r_asm: assembler for many architectures
- r_io: open local files, processes, gdb remote, ..
- r_bin: load ELF/PE/MACH0/CLASS information
- r_util: hexpairs or string to be written
- r_core: run plain radare commands

```
hg clone http://radare.org/hg/radare2-extras
```

rapatch example

This is a hello world example using rapatch in order to transform **/bin/ls** into our code, with import/export reuse.

```
$ cat patch.txt
entry0 {
        printf@alias(${imp.printf});

        main@global(128,128) {
                printf("%d0, 33);
                printf("12345678900);
                printf("hello world0);
                : mov eax,1
                : int 0x80
        }
}
$ cp /bin/ls ls
$ rapatch ls patch.txt
$ ./ls
33
1234567890
hello world
```

Example

Let's patch a simple buggy program

```
static int length(const char *str) {
        char tmp[32];
        strcpy (tmp, str);
        return strlen (tmp);
}

int main(int argc, char **argv) {
        return length ((argc>1)?argv[1]:"");
}
```

If the given string is bigger than the local buffer the stack will be corrupted and the attacker may execute code.

```
$ rapatch bin patch.txt
$ rapatch attach://3842 patch.txt
```

Example: Solutions

We can workaround or fix the issue.. depending on the time or requirements we have to give a solution to the issue:

* change stackframe of function **length**
  - overflow will be still there, but will make the program safe for a standard attack like worms or kiddies do.

* use **strncpy**
  - more complicated patch (code will not fit)
  - a perfect target for rarc2 and rapatch

Solution: Resize stack frame

Changing the stack frame size is a simple and effective solution to change the way the exploit must work.

```
$ r2 -w a.out
[0x8048312]> s sym.readbuf+3
[0x8048312]> pd 3
0x8048312  55      push %ebp
0x8048313  89e5    mov %esp, %ebp
0x8048315  83ec18  sub $0x18, %esp
[0x8048312]> s+3
[0x8048315]> wa sub esp, 0x1024
```

Solution: Using strncpy

The function we have to call in order to make the code secure requires one argument more to define the boundaries of the target buffer.

If we have not enought space to push another argument to the stack have to wrap the execution flow.

Let's see the what we need..

Solution: A nest for our eggs

Where should we put our eggs?

- function paddings
- unused symbols (like init, fini)
- plugins (shared memory)
- injected libraries (LD_PRELOAD)
- mmaped bins on disk
- FMI: read phrack 64

Solution: Hook call

Hooking calls is probably the easiest way to redirect code by patching binaries. Calls or branch+link instructions are constructed by (opcode)(delta). The delta address can be modified in order to branch to our code with 3 byte delta addressing.

A branch to an import can be wrapped patching the **call** opcode and then branching back to the caller address.

Calls store the return address in stack (x86) or in a register on (arm), so we can easily restore the work-flow.

Solution: 2 byte patch (eb)

Another way to get a place for our eggs is by injecting short jumps (2 bytes) in the function prelude. They can branch to near addresses (+/- 129 bytes).

We can replace a 2 byte instructions (mov esp, ebp) in order to redirect to a bigger place where we put our egg.

```
 mov esp, ebp        ->   jmp short ...

$ r2 -w malloc://200000
[0x00001000]> e asm.profile=simple
[0x00001000]> wx eb00 && pd 1
0x00001000  jmp 0x1002 [1]
[0x00001000]> wx eb7f && pd 1
0x00001000  jmp 0x1081 [1]
[0x00001000]> wx eb80 && pd 1
0x00001000  jmp 0xf82 [1]
[0x00001000]> wx ebff && pd 1
0x00001000  jmp 0x1001 [1]
```

Example2: format string

A simple format string vulnerable program:

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

The way patched code should decompile as:

```
printf ("%s", argv[1]);
```

Solution: format string

Hooking the call with our printf trampoline

```
$ cat fmtstr.rasm
.equ printf, 0x8048400
  enter
  lea eax, dummy
  mov [esp+4], eax
  lea esp, dummy
  call printf
  leave
  ret
dummy:
.string "%s"
$ rasm2 -f fmtstr.rasm
```

Example4: boundaries

A famous tramoline boundary vuln in the SMB implementation in Win7, Vista, 2008 (CVE-2009-3103)

The protocol version word (16bits) is used as an index to a pointer table without any check.

This issue was reported as a DoS by *Laurent Gaffie*

`http://g-laurent.blogspot.com/2009/09/windows-vista7-smb20-negotiate-protocol.html`

Later *Ruben Santamarta* published that it was exploitable

`http://blog.48bits.com/2009/09/08/acerca-del-bsod-de-srv2sys`

Example4: boundaries

This is the vulnerable code of the CVE-2009-3103:

```
  movzx eax, word ptr [esi+0xc]
  mov eax, ValidateRoutines[eax*4]
  test eax, eax
  jnz bb_156c9
  mov eax, 0xc0000002
  jmp bb_156cc

bb_156c9:
  push ebx
  call eax

bb_156cc:
  ...
  ret
```

Solution: boundaries

Some languages (like Vala) support {pre,post}conditions.

This is quite close to pseudocode, but allows to define in an elegant way the restrictions for a function to run.

We can wrap the function call by a precondition trampoline which checks and passes the filtered arguments to the destination function.

```
test eax, eax     # bottom limit (<0)
js ${sym.food}
cmp eax, 128      # top limit
ja ${sym.food}
```

Example5: null pointers

Not usually exploitable, but annoying if the program you use crashes because of this.

Fixing it results in missing functionality because of the bug in the target program.

```
cmp $0, 8(%esp)
jz leaveret
```

```
test eax, eax
jz leaveret
```

Some real examples

We are going to analyze the w32 LNK vulnerability and the iOS PDF bug in order to see how they work and find a fix.

W32 LNK bug (CVE 2010-2568)

Exploited by stuxnet. But it was in fact a pretty old bug.. which is respawned every few years :)

The key of this vulnerabilty consists in executing code placed in the constructor of a DLL library which has been specified as an icon of an LNK shortcut file.

The LoadLibraryW of the DLL is triggered by an absolute path in the icon of the crafted LNK file.

`http://www.ivanlef0u.tuxfamily.org/?p=411`

W32 LNK bug (CVE 2010-2568)

The metasploit exploit serves the LNK and the DLL as a randomized file via WebDAV which is displayed as a shared directory service by Windows which can be browsed.

```
$ cat exploit.sh
#!/bin/sh
cd /home/pancake/prg/metasploit
sudo ./msfcli P << EOF
use windows/browser/ms10_046_shortcut_icon_dllloader
set SRVHOST 0.0.0.0
set SRVPORT 80
set PAYLOAD windows/exec
set CMD calc
set LHOST 0.0.0.0
set LPORT 8081
exploit
EOF
```

Open "My PC" on Windows. Go for a shared service and v

LoadLibraryW to be executed by **explorer.exe**

W32 LNK bug (CVE 2010-2568)

To see how the exploit works we can wget the .lnk and .dll files, attach the debugger to the explorer process and wait for LoadLibraryW to go

```
$ r2 -d ‘tasklist | grep explorer.exe | awk ’{print $1}‘
[0x004030d8]> .!rabin2 -rs c:\windows\system32\kernel32.dll
[0x004030d8]> db sym.LoadLibraryW
[0x004030d8]> dc
[0x004030d8]> s eip
[0x7ca78712]>
[0x7ca78712]> pm ds @ esp
0x00f5e9c4 0x0400020
0x00f5e9c8 "Z:\129dd92fg1.dll"
[0x7ca78712]> "wa mov eax,0;ret"
[0x7ca78712]> db-sym.LoadLibraryW
[0x7ca78712]> wc
...
```

iOS PDF bug (CVE 2010-1797)

The site jailbreakme.com exploited it together with an iOS kernel bug which permits priviledge scalation.

The vulnerability is in the freetype library (CFF)

- freetype is GPL, so we can get the fix from the git repo

```
git clone http://git.savannah.gnu.org/cgit/freetype/freetype2.
git diff 236fc8e15a9459d05656013727a1717dbfa425c2
```

```
# in symbol _cff_decoder_parse_charstrings
+ if (decoder->top - stack >= CFF_MAX_OPERANDS)
+   goto Stack_Overflow;
```

iOS PDF bug (CVE 2010-1797)

System libraries and Frameworks in iPhoneOS>3.0 are stored inside the **dyld cache**.

Each ipsw from Apple comes with new dyld cache (~95MB) and each library/binary is built with randomized symbol order.

This makes plain bindiffing impossible at byte level.

We can use **radiff2 -g** to analyze symbols and find differences between the two given binaries.

System wraps all dyld_ calls in order to open files from cache instead of the filesystem, so mach0 loader, gdb and other can still work, but libraries are not in the filesystem.

iOS PDF bug (CVE 2010-1797)

I used to extract the caches of two different iPad devices running 3.2 and 3.2.1 respectively:

```
$ rabin2 -A ../dyld_shared_cache_armv7
 000 /System/Library/PrivateFrameworks/StoreServices.
     framework/StoreServices arm_32 (Unknown arm subtype)
 001 /System/Library/Frameworks/CFNetwork.framework/
     CFNetwork arm_32 (Unknown arm subtype)
 002 /usr/lib/libarchive.2.dylib arm_32 (Unknown arm ..
 ...

$ rabin2 -x dyld_shared_cache_armv7 libCGFreetype.A.dylib
```

iOS PDF bug (CVE 2010-1797)

Checking for the symbols of each library we get the same number of functions

```
$ rabin2 -s libCGFreetype.A.dylib.3.2 > /dev/null
864 symbols
$ rabin2 -s libCGFreetype.A.dylib.3.2.1 > /dev/null
864 symbols
```

Symbol order in libCGFreetype looks not randomized. But you may find this in other libs like UIKit.

```
$ rabin2 -s UIKit-3.2 |head | awk -F = '{print $2" "$9}'
0x31602170 offset _CGAffineTransformFromString
0x315e6adc offset _CGPointFromString
0x3160234c offset _CGRectFromString
...
$ rabin2 -s UIKit-3.2.1 |head | awk -F = '{print $2" "$9}'
0x3022e0d8 offset _CGAffineTransformFromString
0x30212a44 offset _CGPointFromString
0x3022e2b4 offset _CGRectFromString
...
```

iOS PDF bug (CVE 2010-1797)

As seen in the git, the vulnerability is found in the _cff_decoder_parse_charstrings symbol which incorrecly check the boundaries of the arguments in the VM of the fonts.

Cydia published a PRELOAD patch based on the MobileSubstrate framework which consists on a system-wide library preloading that offers many ways to extend or manipulate the iOS interface and functionality. (sms helper, etc..)

The patch from cydia is not opensource, but analyzing it with r2 is as simple as expected.

iOS PDF bug (CVE 2010-1797)

Let's unpack the debian package from cydia..

```
$ apt-get install com.saurik.iphone.cve-2010-1797_11.0.3245
$ cd /var/cache/apt/archives
$ ar x com.saurik.iphone.cve-2010-1797_1.0.3245-1_iphoneos-arm
$ tar xzvf data.tar.gz
./Library
./Library/MobileSubstrate
./Library/MobileSubstrate/DynamicLibraries
./Library/Mobile.../.../PDFPatch_CVE-2010-1797.dylib
./Library/Mobile.../.../PDFPatch_CVE-2010-1797.plist
```

The **dylib** is the mobilesubstrate plugin that fixes the bug.

iOS PDF bug (CVE 2010-1797)

The **PDFPatch_CVE-2010-1797.dylib** library is the mobilesubstrate plugin that patches the freetype library in memory to fix the bug

```
$ cd Library/MobileSubstrate/DynamicLibraries/
$ r2 *.dylib
[0x00000dfc]> s sym.__ZL13_MSInitializev
[0x00000f08]> pd
    0x00000f08    push {r7, lr}
    0x00000f0c    ldr r0, [pc, #12]
    0x00000f10    add r7, sp, #0
    0x00000f14    add r0, pc, r0
    0x00000f18    bl imp.__dyld_register_func_for_add_image
    0x00000f1c    pop {r7, pc}
[0x00000dfc]> s imp.__dyld_register_func_for_add_image
...
```

Questions?

Ideas, questions?

jan seme li wile sona? o toki!
wile e toki suli ni li pona tawa jan :)

Thanks for listening!