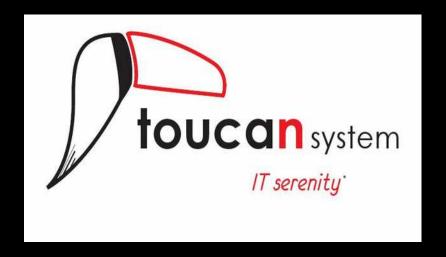
# Generic exploitation of invalid memory writes

Jonathan Brossard CEO – Toucan System



jonathan@ toucan-system.com



## Agenda

- Introduction
  - Memory permissions
    - Taxonomy of invalid memory access
  - Determining the type of error
- The case of invalid writes

#### Introduction

- Maybe you did some fuzzing and found some bugs?
- Welcome in 2010: exploitation is more difficult than finding trivial bugs.
- The goal of this talk is to detail (and automate!) the steps to successful exploitation of invalid memory writes.

This is the key to modern exploitation...

Under x86 : permissions are set at MMU (+ Kernel) level using three flags :

R: read permission

W: write permission

X : execution permission

5 years back, everything was in fact +X!!

Where to get my shellcode executed in memory ??

Alternatively (when not using shellcode): where to return in memory to execute something usefull (think ROP).

What is or not executable depends on:

- your cpu (NX flag) + BIOS Config.
- your kernel (PaX, PAE...) and more globally your toolchain.
- the dynamic loader (shared libs).

#### DEMO

Using readelf and /proc/pid/maps to check what permissions should be.

Using paxtest to verify what is <u>actually</u> executable or not.

# Taxonomy of invalid memory access

What's the impact of this bug?

```
int main(int argc, char **argv){
          printf(argv[1]);
          return 0;
}
```

### Exemple 1

Truth is: we <u>can't</u> know from C source: we have to check at instruction level to see if the c library was actually built with a write primitive...

#### Segfault at...

mov DWORD PTR [eax], edx

eax 0x4000e030

ecx 0xbffff848

edx 0x0

ebx 0x40199ff4

=> Invalid write.

#### (more complex) exemple:

fld QWORD PTR [eax+0x8]

=> Read? Write? Both??

#### DEMO

Triggering the bug in xpdf

Determining the error type of complex bugs using valgrind

## Summarizing

At instruction level, an invalid memory access can be of three kinds:

- invalid read
- invalid write
- invalid exec

#### Exploiting invalid exec

Trivial if jump location is user controled.

eg:call[eax]

Exploitation strategy:
Have eax pointing to our shellcode

#### Exploiting invalid exec (2/2)

#### Fairly rare in userland

Used to be a major problem in kernel land (invalid Null ptr dereference in exec mode from kernel land). First page not mappable without privs since kernel 2.6.23

## Exploiting invalid reads

Cannot hijack control flow directly:(

#### **Exploitation strategy:**

- Either use no memory corruption (eg : information leakage)
- Or trigger a bug latter in the code (either invalid exec or invalid write)
- ... Or it is just plain non exploitable :(

#### Exploiting invalid writes

This is the meat!

#### **Exploitation strategy:**

- Either overwrite important data (eg: tast\_struct->uid in kernel land)
- Or try to hijack the control flow

## The case of invalid writes: different levels of control

#### X86 allows accessing either:

- 8 bytes
- 4 bytes (most instructions)
- 2 bytes
- 1 byte

## The case of invalid writes: different levels of control

- If both destination and value are fully user controled: overwrite.dtors, liked list in at\_exit, function pointers... (ideal case such as missing format strings, easy!)
- If only destination is user controled: attempt a pointer truncation on a given function pointer.

#### Problem is...

Out of 3Gb of user land, how to find a suitable function pointer to overwrite or truncate?

### Exploitation strategy

- Dump the content of every section of the binary
- Parse the +W ones
- Check for pointers to +X locations
- => exhaustive, reduced list of potential candidates.
- Then chose one based on actual binary constraints (will truncation point to a user controled location?)

#### DEMO

Triggering an invalid write in Opera

Using livedump to chose potential candidates

## Memory alignement

Most Intel instructions only allow reads/ writes on aligned boundaries.

Typically: 4b aligned memory

## (Common) Worst case scenario

If only 0x00000000 can be written, in 4b aligned memory locations...

=> Then pointer truncation won't be possible on 4b aligned sections :-(

#### Naive exploitation strategy

- Trigger the bug
- set a signal handler for signal 5 which re enables single stepping
- set the « trap » cpu flag
- => Single step until exit and monitor unaligned reads/writes.
- => Slow, painfull, per thread :(

# Elite exploitation strategy with livedump

- Trigger the bug
- set the « align » cpu flag
- setup a signal handler for SIGBUS
- => every read/write to unaligned memory will be trapped and give potential truncation candidates

#### DEMO

## Using livedump to detect unaligned memory access

## Thank you for coming

#### Questions?



