C Applications Debugging

Adrien Kunysz

August 4, 2012

Outline

Outline

What we will cover

- userland only
- understanding how a process works
- C and x86 assembly basics
  - pointers!
- gdb essentials
  - what to do with a core
  - understanding backtraces
  - disassembling functions

What we will not cover

- kernel stuff (vmcore, crash,. . .)
- swapping and how virtual memory works
- multithreading
- dynamic debugging (valgrind, SystemTap,. . .)
- malloc debuggers (valgrind, ElectricFence,. . .)
- ELF, dynamic loader,. . .
- system calls internals
- PAE and other 32 bits specific hacks

Outline

Disclaimer

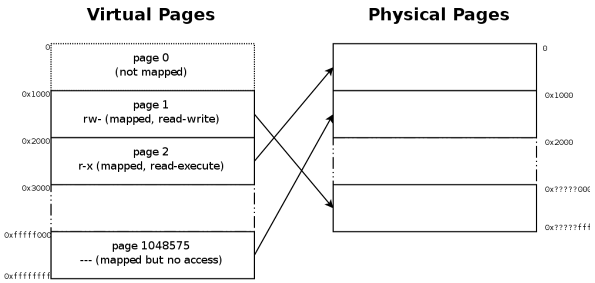
- everything in this section is wrong
- or at least greatly simplified
- but it's fine as long as you don't try to debug kernel issues
- pay no attention to the man behind the curtain (for now)

Basic facts about memory

- this is about 32 bits x86 but similar for other archs
- this is about a single threaded userland process
- memory is just a 2^{32} bytes (4G), 1 dimension array
- splitted into pages

What's a page?

- 2^{12} bytes (4K), 1 dimension array
- access rights for each page: read, write, execute
- can be mapped (backed up by actual memory) or not
- you have to ask kernel to map pages for you
  - `mmap()`, `brk()`, `sbrk()`
  - usually done within `malloc()`, `calloc()`, `realloc()`,...



What’s a segmentation fault?

A segmentation fault is a signal `SIGSEGV` sent by kernel to a process doing something wrong.

- accessing unmapped page
- wrong access rights (writting to read-only page,. . .)

This will write a full copy of the process memory to a file (core).

What's a pointer?

A pointer is a 32 bits value which is an address pointing to the N^{th} cell of our memory array.

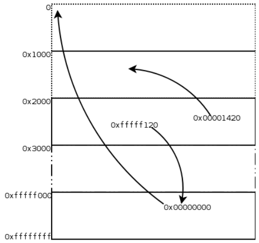
- typically pointing to the beginning of a specific data structure
- special "invalid pointer" value: `NULL`
- pointer fun with Binky!

This is document 104 in the Stanford CS Education Library. Please see `http://cslibrary.stanford.edu/` for this and other free educational materials. Copyright Nick Parlante 1999.

Dangling pointers

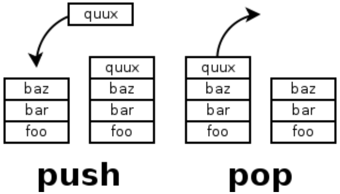
- pointing to unmapped page $\rightarrow$ segfault
- pointing to mapped page but wrong data $\rightarrow$ madness

  ... and probably segfault a few thousands instructions later
- the segfault only happens when trying to *access* the unmapped page



What’s a stack?

- push data on the top
- pop them back from the top in reverse order
- let you store things you will need to resume a task you interrupted



How is the stack used?

- you push a context that you will restore later
- one stack per thread
- some part of the memory is used for that stack
- stack grows too large → overwrites random data, access unmapped memory
- need to keep reference to the top of the stack: stack pointer (ESP on x86)

TODO: stack graphic here

```
int foo(int x) {
        int y;
        int z;
        /* ... */
        z = z + bar(x + y);
        /* ... */
        return z
}
```

Outline

C programming language crash course

This presentation will not make you a C ninja in one hour.

This presentation will not cover:

- how to write C code
- how to fix existing C code
- advanced C tricks

After this presentation, you should have some idea of:

- how C works
- how to read basic C code
- how to find the area where a problem is located
- where to look to figure out the more advanced stuff

What is C?

- low level programming language
- full control of your 2^{32} bytes memory array
  - good for system programming (kernel,...)
  - not so good for application programming
- very easy to make stupid, hard to find mistakes
- not much fancy features
- old legacy that is not going to die any time soon
- "everything" is written in C at some level
  - that includes your Python interpreter
- there are a few different C dialects

K&R	the original one, mostly unused nowadays
ANSI	first standardization, the most common
C99	latest standard, some nice new features but not used a lot

C syntax: basics

- functions
- control structures: `if`, `else`, `while`,...
- comparison operators: `==`, `!=`, `<`, `>`, `<=`, `>=`
- assignation operator: `=`

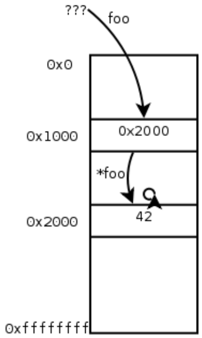
```c
int n = 1;
while (n < 10)
{
        n = square(n);
}
```

```c
int square(int x)
{
        int y;
        if (x == 0) {
                y = 0;
        } else {
                y = x*x;
        }
        return y;
}
```

C syntax: pointers

```
int n;    // declaring integer
int *foo;   // declaring pointer
            // to integer

foo = &n;   // making foo point to n
*foo = 42;   // like n = 42
```



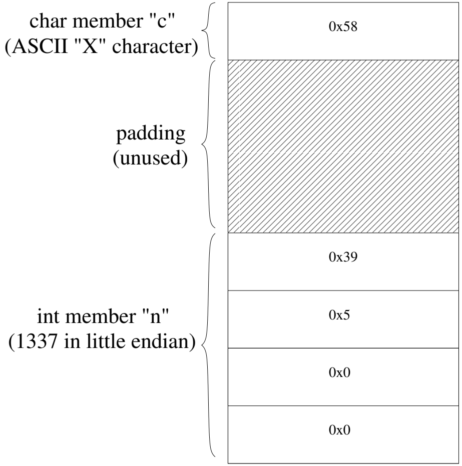
foo = 0x2000
*foo = 42
&foo = 0x1000

C syntax: struct

- customized data types
- let you define complex aggregate of data

```
struct foo_t {
        char c;
        int n;
};
struct foo_t foo;
struct foo_t *p;
struct foo_t *p2;

foo.n = 1337;
p = &foo;
p->c = 'X'; // (*p).c = 'X'
p2->n = 42; // segfault (maybe)
```



Dynamic memory management

- kernel provides low level API (system calls)

mmap	map some pages
munmap	unmap some pages
brk	map some more page at the end of this area

- libc provides higher level API based on top of system calls

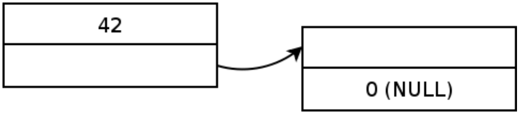
malloc	give me some memory (any size)
free	you can get this memory back
realloc	resize this piece of memory

- applications usually use libc
  - system calls are slower than library calls
  - libc people are smarter than application developers
- `sizeof`
  - C operator
  - evaluated at compile time
  - gives size of its argument
  - usually used to tell `malloc` how much memory we want
- RTFM: `man {malloc,mmap,brk}`

Practical example: a linked list

```
struct list_t {
      int x;
      struct list_t *next; // not recursive, this
                           // is just a 32 bits value
};

struct list_t *ll = malloc(sizeof(struct list_t));
ll->x = 42;
ll->next = malloc(sizeof(*ll));
ll->next->next = NULL;
free(ll);
free(ll->next);   // segfault (maybe)
```



Working with source RPM

- How to use a source RPM?
  - `rpm -ivh foobar.src.rpm`
  - `rpmbuild -bp /usr/src/redhat/SPECS/foobar.spec`
  - `cd /usr/src/redhat/BUILD/foobar/`
- The `/usr/src/redhat/` directory:

BUILD	unpacked and patched sources (after -bp)
RPMS	generated rpm files
SOURCES	source and patch files
SPECS	package metadata (spec files)
SRPMS	generated src.rpm files

- RTFM: `man rpmbuild`

Reading C

- trying to understand a large program all at once is useless
- focus on what you need to understand
- look for the error message and work from there
- tools
  - find, grep, vi,. . .
    - available everywhere
    - you already know how to use them
  - cscope
    - find all the callers of a given function
    - jump to definition of a function or struct
    - slow to startup on large sources but pretty convenient

Let's do it!

- find an application error message
- retrieve the source
- look up the error
- follow the code that triggered the error
- a simple, practical example: rhbz497874

RTFM

- ISO/IEC 9899: the C99 standard (final draft freely available)
- Single UNIX Specification: a superset of the POSIX API
- Standard C: a readable reference
- `man $ANYFUNCTION`

Outline

x86 Assembly Language crash course

- (very) low level programming language
  - human readable machine code
- "nobody" writes in assembly anymore
  - very tedious (computer's job)
  - the compiler is better at it than you
- you only read it to figure out what went wrong when the your debugger can't figure it out itself
- two main syntaxes in use

AT&T	is used by the GNU assembler
NASM	is for Netwide Assembler

Quick comparison

- This is C (from Samba's `inotify_handler()`):

```
if ((ioctl(in->fd, FIONREAD, &bufsize) != 0) &&
    (errno == EACCES)) {
```

Quick comparison

- This is C (from Samba's `inotify_handler()`):

```
if ((ioctl(in->fd, FIONREAD, &bufsize) != 0) &&
    (errno == EACCES)) {
```

- This is x86 assembly:

```
mov     %eax,-0x28(%ebp)
lea     -0x10(%ebp),%eax
mov     %eax,0x8(%esp)
movl    $0x541b,0x4(%esp)
mov     -0x28(%ebp),%edx
mov     0x4(%edx),%eax
mov     %eax,(%esp)
call    0x7b35e8 <ioctl@plt>
test    %eax,%eax
jne     0x9ecc74 <inotify_handler+164>
mov     -0x10(%ebp),%edx
test    %edx,%edx
jne     0x9ecc96 <inotify_handler+198>
```

Quick comparison continued

This is x86 machine code (in one byte hex chunks for readability):

```
0x89 0x45 0xd8 0x8d 0x45 0xf0 0x89 0x44 0x24 0x08
0xc7 0x44 0x24 0x04 0x1b 0x54 0x00 0x00 0x8b 0x55
0xd8 0x8b 0x42 0x04 0x89 0x04 0x24 0xe8 0xc5 0x69
0xdc 0xff 0x85 0xc0 0x75 0x4d 0x8b 0x55 0xf0 0x85
0xd2 0x75 0x68
```

x86 registers

- CPU can only directly access register
- each register contains a 32 bits value
  - can be a value
  - can be an memory address (pointer)
- different types of register:
  - 8 general purpose registers
  - 6 segment registers (CS, DS, SS, ES, FS, GS)
  - 1 EFLAG register (various CPU status info)
  - 1 instruction pointer (EIP)

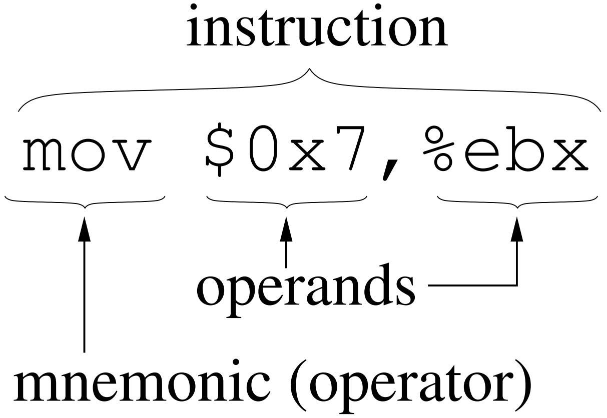
x86 general purpose registers

- EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- used for about anything the compiler feels like
- except for

EBP	base pointer: address of the current stack frame
ESP	stack pointer: address of the top of the stack

TODO: stack picture here

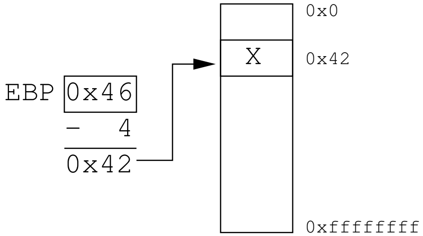
x86 AT&T assembly syntax basics



- registers are prefixed with %
- integer constants are prefixed with $
- everything else after the last operand is ignored (comment)

x86 AT&T assembly syntax memory addressing

- `(%eax)` is for dereferencing EAX (kind of like `*eax` in C)
- `-4(%ebp)` means `*(ebp-4)`



- the index is in bytes (so, −4 is 32 bits earlier)
- more advanced dereferencing tricks you shouldn't care too much about:
  - `%ds:-42(%ecx,%edx,2)` is kind of like `*(ds+(ecx+edx*2-42))`

x86 AT&T syntaxe: operands size

- "vanilla" operators use the register size to determine the operands length
- but you can use a different length by using specific suffixes:

b	bytes
l	long

TODO: this part is not complete

Outline

NOT WRITTEN YET