hack.lu CTF 2013 - ELF 400 - Making Of

sqall's blog

date_format_entry

hack.lu CTF 2013 - ELF 400 - Making Of

作者 **sqall** 分类 **CTF**, **security**, **Tutorials**

回复 **(0)**
引用 **(0)**

Some guys on IRC asked me how I made the ELF challenge of the hack.lu CTF 2013. If I just used a HEX editor or some other tool. So I decided to write this little "making of".

First to the challenge itself. The text for the challenge was:

> We encountered a drunk human which had this binary file in his possession. We do not really understand the calculation which the algorithm does. And that is the problem. Can you imagine the disgrace we have to suffer, when we robots, based on logic, can not understand an algorithm? Somehow it seems that the algorithm imitates their masters and behaves …. drunk! So let us not suffer this disgrace and reverse the algorithm and get the correct solution.

It still can be downloaded under http://h4des.org/source/challenges/hacklu2013/hacklu2013_elf. There was a problem under certain Linux Distributions like Ubuntu, Xubuntu and ArchLinux. When they were used, the calculation of the flag was wrong. This announcement was given during the contest to clarify things:

> Ok I think we got it (thanks to Happy-H from Team ClevCode). Ubuntu introduced a patch to disallow ptracing of non-child processes by non-root users. This changes the calculated value. So when you use Ubuntu you should work as root. The other distributions should not be affected.
>
> Anyway, I created a VM where the executable works just fine: http://h4des.org/source/challenges/hacklu2013/hacklu2013_elf.ova (User: elf:elf and root:root)

The VM is still downloadable via the link, if you want to solve the challenge by yourself. A really good write-up can be found here: http://bases-hacking.org/elf-hacklu2013.html.

Ok, now let us take a look at the source of code of the binary:

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

navigation

old projects (h4des.org) (german)
impressum (german)
h4des.org root certificate (german)
Twitter: @sqall01

快速搜寻

>

站点日历

October '13

Mon	Tue	Wed	Thu	Fri	Sat	Sun
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

文章归档

October 2013
September 2013
August 2013
新文库...
旧文库...

kategorien

blog
coding
CTF
eleusis
fail
fun
kastalia
linux
Debian
netculture
OpenSource
Politik
RealLife
security
Tutorials
Ueberwachung
virtualization
vdi
windows

全部类别

creative commons

```
// global variable for anti ptrace
unsigned int anti_ptrace_var = 10;
char * phase2Const =
"fluxFluxfLuxFLuxflUxFlUxfLUxFLUxfluXFluXfLuXFLuXflUXFlUXfLUXFLUX";



// this function checks if ptrace is used for debugging
void anti_ptrace(void) {
	pid_t child;

	// check if LD_PRELOAD is set
	if(getenv("LD_PRELOAD"))
		anti_ptrace_var++;

	// returns child PID in the parent and 0 in the child
	child = fork();

	// check if process is child or parent
	if(child) { // => process is parent
		// wait until child is terminated
		int status;
		wait(&status);

		// check if child could not attach to parent with ptrace => ptrace
already in use
		if(status != 0) {
			// also use sleep when child could not attach to parent
			sleep(1);
			anti_ptrace_var++;
		}

	}
	else { // => process is child
		pid_t parent = getppid();

		// try to attach to parent
		if (ptrace(PTRACE_ATTACH, parent, 0, 0) < 0)
			// exit with an other status code to signal parent that ptrace
could not be used
			exit(1);

		// wait a second to give the OS time between attaching and
detaching
		sleep(1);

		// detach to parent
		ptrace(PTRACE_DETACH, parent, 0, 0);

		// exit child
		exit(0);
	}
}


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


	// garbage variable to get more space in the stack for the modification
	char * placeholder[208];


	// check if ptrace or ld_preload is present
	anti_ptrace();
```

```
	// check if arguments count is correct
	if(argc != 2) {
		printf("Usage: %s <flag>\n", argv[0]);
		exit(0);
	}

	puts("Calculating phase 1 ...");

	char *inputFlag = argv[1];
	char *phase1 = (char *)malloc(strlen(inputFlag));

	unsigned int i, j;
	for(i = 0; i < strlen(inputFlag); i++) {
		phase1[i] = inputFlag[(i - anti_ptrace_var) % strlen(inputFlag)];
	}

	sleep(1);

	puts("done\n");


	// increase anti_ptrace_var => it is not possible to use a constant to
reverse this algorithm
	anti_ptrace_var++;


	// calculate some stuff in order to get more space in the stack for the
modification
	for(i = 0; i < 208; i++) {
		placeholder[i] = (char)0x41;
	}

	char *phase2 = (char *)malloc(strlen(inputFlag));

	puts("Calculating phase 2 ...");

	for(i = 0; i < strlen(inputFlag); i++) {
		phase2[i] = phase2Const[i] ^ anti_ptrace_var ^ phase1[i];
	}

	sleep(1);

	puts("done\n");


	// increase anti_ptrace_var => it is not possible to use a constant to
reverse this algorithm
	anti_ptrace_var += 3;


	char *finalPhase = (char *)malloc(strlen(inputFlag));
	for(i=0; i < strlen(inputFlag); i++) {
		finalPhase[i] = (unsigned char)(anti_ptrace_var & 0xFF);
	}


	// calculate some stuff in order to get more space in the stack for the
modification
	for(i = 0; i < 208; i++) {
		placeholder[i] = (char)0x42;
	}

	// calculate some stuff in order to get more space in the stack for the
modification
	for(i = 0; i < 208; i++) {
```

```
                placeholder[i] = (char)0x46;
        }


        for(i = 0; i < 3; i++) {
                printf("Calculating phase %u ...\n", (i+3));

                for(j = 0; j < strlen(inputFlag); j++) {

                        finalPhase[j] = phase2[j] ^ finalPhase[j] ^ (unsigned
int)phase2Const[ (j+i+anti_ptrace_var) % strlen(inputFlag) ];
                }

                sleep(1);

                puts("done\n");
        }


        // calculate some stuff in order to get more space in the stack for the
modification
        for(i = 0; i < 208; i++) {
                placeholder[i] = (char)0x45;
                placeholder[i] = (char)0x43;
                if(placeholder[(i+3) % 208] == 0x41) {
                        placeholder[(i+4) % 208] = (char)0x53;
                }
        }
        // calculate some stuff in order to get more space in the stack for the
modification
        for(i = 0; i < 208; i++) {
                placeholder[i] = (char)0x43;
                if(placeholder[(i+3) % 208] == 0x41) {
                        placeholder[(i+4) % 208] = (char)0x53;
                }
                if(placeholder[(i+3) % 208] == 0x42) {
                        placeholder[(i+4) % 208] = (char)0x53;
                }
        }


        unsigned int solution1 = 0;
        unsigned int solution2 = 0;
        unsigned int solution3 = 0;
        unsigned int solution4 = 0;

        for(i = 0; i < 4; i++) {
                solution1 = solution1 | (((unsigned int)finalPhase[i] & 0xFF) <<
(8*i));
        }


        for(i = 0; i < 4; i++) {
                solution2 = solution2 | (((unsigned int)finalPhase[i+4] & 0xFF) <<
(8*i));
        }

        for(i = 0; i < 4; i++) {
                solution3 = solution3 | (((unsigned int)finalPhase[i+8] & 0xFF) <<
(8*i));
        }

        for(i = 0; i < 4; i++) {
                solution4 = solution4 | (((unsigned int)finalPhase[i+12] & 0xFF) <<
(8*i));
        }
```

```
        }


        if(solution1 == 0x58326011
        && solution2 == 0x22516561
        && solution3 == 0x5e6b6266
        && solution4 == 0x556e454b) {
                puts("Flag correct!");
        }
        else {
                puts("Flag wrong!");
        }


        return 0;

}
```

The code got two functions: the "main" and the "anti_ptrace" function. The "main" function first calls the "anti_ptrace" function and then starts some calculation with the given argument. After the calculation is done, it is checked with some fixed values. When the values are equal to the calculation, the string "Flag correct!" is printed. So the given argument have to be the flag. When we take a closer look at the calculation, we see that it works only byte-wise with some shifting and xoring. So it can be reversed and due to the fact that it is checked with four 4 Byte values, we can assume that the flag has 16 Bytes (and is also printable because the flag should be submitted to the scoreboard of the CTF afterwards). The calculation uses the global variable "anti_ptrace_var" which is initialized with 10. Let us keep that variable in mind.

Now we take a look at the "anti_ptrace" function. It tests if the environment variable "LD_PRELOAD" is set and increases the "anti_ptrace_var" variable accordingly. Also it checks if a child process can attach to the parent process via ptrace. When the child process can not attach, ptrace is already used by a debugger like GDB or the IDA debugger. So the "anti_ptrace_var" variable is increased when the attaching fails. Long story short, this function just tries to detect some debugging techniques and changes the "anti_ptrace_var" variable accordingly. And when the "anti_ptrace_var" variable is changed, then the calculation of the flag is altered too. As a result, a wrong flag is calculated.

This would be very easy to detect and patch out. So how can we hide this anti debugging function? The idea for this came when I tried to learn some things about the ELF format. I wrote a parser library in python to understand

the format and soon it got some manipulation features. I published the incomplete library I used at github after the CTF was finished. An ELF binary contains "sections" and "segments". Most analysis tools interpret only the "sections" of the binary (like IDA when you do not change the default settings). But the Linux loader uses the "segments" to load the binary and execute it. "Sections" itself are not important to execute an ELF binary and are optional. They are just used for debugging purposes. So the idea was to manipulate the binary in such a way, that the code resides in a "segment" but not in a "section".

I wrote two different assembler functions for anti-debugging purposes. The first one:

```
push   ebp
mov    ebp,esp
sub    esp,0x28
```

```
call   gotRepairDataMemoryAddr
mov    DWORD PTR [esp], ldPreloadStringMemoryAddr
call   8048450 <getenv@plt>
test   eax,eax
je     logical_label1
mov    eax, antiPtraceVarMemAddr
add    eax,0x1
mov    antiPtraceVarMemAddr, eax
logical_label1:
call   80484a0 <fork@plt>
mov    DWORD PTR [ebp-0xc],eax
cmp    DWORD PTR [ebp-0xc],0x0
je     logical_label2
lea    eax,[ebp-0x14]          ; only reached by parent
mov    DWORD PTR [esp],eax
call   8048440 <wait@plt>
mov    eax, antiPtraceVarMemAddr       ; increase anti_ptrace variable
without detecting ptrace
add    eax,0x1
mov    antiPtraceVarMemAddr, eax
mov    eax,DWORD PTR [ebp-0x14]        ; evaluate wait() status code
(ptrace found?)
test   eax,eax
je     logical_label4
mov    DWORD PTR [esp],0x1
call   8048430 <sleep@plt>
mov    eax, antiPtraceVarMemAddr
add    eax,0x1
mov    antiPtraceVarMemAddr, eax
jmp    logical_label4
logical_label2:         ; only reached by child
call   80484b0 <getppid@plt>
mov    DWORD PTR [ebp-0x10],eax
mov    DWORD PTR [esp+0xc],0x0
mov    DWORD PTR [esp+0x8],0x0
mov    eax,DWORD PTR [ebp-0x10]
mov    DWORD PTR [esp+0x4],eax
mov    DWORD PTR [esp],0x10
call   80484c0 <ptrace@plt>
test   eax,eax
jns    logical_label3
mov    DWORD PTR [esp],0x1
call   8048480 <exit@plt>
logical_label3:

mov    DWORD PTR [esp],0x1
call   8048430 <sleep@plt>
mov    DWORD PTR [esp+0xc],0x0
mov    DWORD PTR [esp+0x8],0x0
mov    eax,DWORD PTR [ebp-0x10]
mov    DWORD PTR [esp+0x4],eax
mov    DWORD PTR [esp],0x11
call   80484c0 <ptrace@plt>
mov    DWORD PTR [esp],0x0
call   8048480 <exit@plt>
logical_label4:
nop
leave
```

This is basically the same as the "anti_ptrace" function. The only two differences are, that this function calls the address "gotRepairDataMemoryAddr" first (I explain later what this does) and it always increases the "anti_ptrace_var" variable by one. This means that this function

changes the "anti_ptrace_var" variable not only if a debugger was found and have to be considered in the calculation of the correct flag.

The second function:

```
push eax
push ecx
call   gotRepairDataMemoryAddr
mov eax, len(putsData)-1
puts_start_label:
mov cl, [eax+ putsDataMemoryAddr ]
cmp cl, 0xcc
jne puts_logical_label1
push eax
mov    eax, antiPtraceVarMemAddr
eax,0x1
antiPtraceVarMemAddr, eax
pop eax
puts_logical_label1:
cmp eax, 0
je puts_end_label
dec eax
jmp puts_start_label
puts_end_label:
mov eax, len(strlenData)-1
strlen_start_label:
mov cl, [eax+ strlenDataMemoryAddr ]
cmp cl, 0xcc
jne strlen_logical_label1
push eax
eax, antiPtraceVarMemAddr
eax,0x1
antiPtraceVarMemAddr, eax
pop eax
strlen_logical_label1:
cmp eax, 0
je strlen_end_label
dec eax
jmp strlen_start_label
strlen_end_label:

pop ecx
pop eax
```

This function calls the "gotRepairDataMemoryAddr" memory address first, then searches for 0xcc bytes at a specific memory range and increases the "anti_ptrace_var" variable every time a 0xcc was found. 0xcc bytes are used for software breakpoints in debugger.

These are basically the two anti debugging functions we want to hide in the memory with the idea explained above. But now the question: how can we execute these functions without anyone to notice?

For this we can use the GOT (Global Offset Table). The GOT stores the memory addresses to external functions like printf(). These memory addresses are used to call the external function. But when the binary is executed and an external function is called the first time, the memory address of the external function is not known. In order to find the memory address of the external function dynamically during the execution the PLT (Procedure Linkage Table) is used. Normally, the GOT entry for an external function holds the memory address of the PLT entry +6 (under x86). When we call an external function (for example with "call printf") the control flow

jumps to the PLT which then will jump to the memory address that is stored in the GOT for this external function. If the external function is called for the first time, then we will jump to the same PLT entry +6 and some linking black magic will then happen that sets the correct memory address in the GOT entry and executes the external function. Now when the same external function is called again, the PLT entry will directly jump to the GOT entry value (which was correctly set to the memory address of the external function by the linking black magic). This was a short description how the dynamic linking of external functions works under Linux. There is plenty of material for more details if you are interested. But to give an example, here a copy & paste from GDB:

```
;The PLT for abort:
0x80495b8 <abort@plt>:  jmp    DWORD PTR ds:0x8060098
0x80495be <abort@plt+6>:        push   0x0
0x80495c3 <abort@plt+11>:       jmp    0x80495a8

;The initial GOT entry for abort:
0x8060098 <abort@got.plt>:      0x080495be

;Some arbitrary call of abort:
0x804d8d1 call   80495b8 <abort@plt>

;After abort() is called the first time, the GOT entry for abort will change:
0x8060098 <abort@got.plt>:      0x07044588
```

Ok back to the challenge. The memory address of the initial GOT entries are stored in the ELF binary (the ones that point the the PLT). So we can replace them with the memory addresses of our code and a call to an external function like printf() would execute our code. Analysis tools often do not show these addresses (or the analyst will usually not check them). So it is hidden pretty well when static analysis is used. But to also execute the code of the correct external function we have to jump back to the PLT +6 memory address of the hijacked external function. The problem with this is, that the dynamic linking process will overwrite our hijacked GOT entry and the code

can only be executed when the external function is called the first time.

To solve this problem the two functions explained above call first the "gotRepairDataMemoryAddr" memory address. The code that resides at this address looks like this:

```
push eax
mov eax, putsDataMemoryAddr
mov [putsGotMemoryAddr], eax
mov eax, printfDataMemoryAddr
mov [printfGotMemoryAddr], eax
mov eax, mallocDataMemoryAddr
mov [mallocGotMemoryAddr], eax
mov eax, strlenDataMemoryAddr
mov [strlenGotMemoryAddr], eax
pop eax
ret
```

This function overwrites all hijacked GOT entries with the memory addresses to our code. In the ELF challenge the external functions "puts", "printf", "malloc" and "strlen" are hijacked. So this function overwrites the GOT entries of these four external functions with the memory addresses to our code. But this is done during the execution of our code. This means that the dynamic linking process will overwrite the GOT entry of the external function

dynamic linking process will overwrite the GOT entry of the external function just executed afterwards. As a result, the same hijacked function can not be called twice in a row. For example if strlen() is executed and after that strlen() is executed again, our code will only be executed the first time. If the execution is strlen(), malloc(), strlen(), then our code for strlen() will be executed twice because the call of malloc() will restore the GOT entry to our hidden code.

The ELF challenge uses the code that checks for 0xcc bytes for "malloc" and "printf". For "puts" and "strlen" the anti ptrace code is used. The memory ranges that are checked by "malloc" and "printf" are the memory addresses of the code of "puts" and "strlen". This means that "malloc" and "printf" check if a software breakpoint was set in the code of "puts" and "strlen". In order to increase the value of the "anti_ptrace_var" variable every time "malloc" and "printf" are called, a 0xcc byte was placed manually in the "puts" and "strlen" code.

Ok, we finally have all our code and ideas together. We have to keep in mind that every call of our hijacked functions changes the "anti_ptrace_var" variable during the calculation. The last thing that is missing is a way to inject our code. I wrote this script that uses my python manipulation library and injects our code. The script uses the free space in memory of the executable "segment" (which exists because of the padding) and injects the new code there. In order to work correctly, the "segments" and "sections" in the ELF binary file itself have to be moved around (and the memory addresses of all "segments"/"sections" have to stay the same).

After that, we have to adjust the fixed values in the binary that are used to check if the flag is correct. This means we have to consider the changes our injected code does. For this we can use an arbitrary HEX editor and the challenge is finished (during the creation I just used the existing printf() to get the correct values).

One problem that I encountered during the tests was, that the injected code got a segmentation fault while executing "push eax" or "call

gotRepairDataMemoryAddr". The only explanation I had was, that the stack was not writeable at this position. I do not know why or how I can change the stack position in the ELF binary. But to solve this problem I have to add some pointless calculations in the C code. For someone who wants to solve this challenge it looks like obfuscation, but it was only inserted to move the stack around.

引用

引用此文章特定的网址

没有引用

回复

回复显示方式 (直线程 | 分线程)

没有回复

新增回复

名称

电邮

网址

回复 [最高层]

到

回复

像 :-) 和 ;-) 等标准的表示表情色彩的字符串会被转换成图片形式
电子邮件地址将不会被显示，而仅将被用于发送电子邮件通知

为了阻止机器人提交垃圾回复，请在相应的文本框中输入你在下面的图片中所看到的字符串。只有在你输入的字符串和图片中的字符串吻合的情况下，你的回复才能被成功提交。请确认你的浏览器支持、并且已经开启了cookies功能，否则的话，你的回复无法被正确地验证。

请输入上面图片中所显示的字符:

记录资料

订阅这篇文章

发布的回复需要管理员审核

发送回复 预览

首页 - 最高层