

# Securing Services: With Emphasis on Exploitation and FreeBSD Jails

Anish Mistry

## Introduction

In a computer world where security and performance has to co-exist and where software has to eventually run on an operating system and in an environment that must allow it to do useful work. Being able to run your service to a hostile outside world is indeed a daunting task. Over 2000 years ago Sun Tzu said, “If you know the enemy and know yourself, you need not fear the result of a hundred battles.” (11) We must first know the methods of the enemy, their weapons, methods, and goals. We must know ourselves. For Unix based operating systems there are various methods and levels of restrictions that may be put into place to isolate and reduce the impact of a compromised or insecure service. Primarily these are provided by software checks, unprivileged users, chroot, and FreeBSD jails, which are also know as zones in Solaris 10 (4).

## Know your enemy...

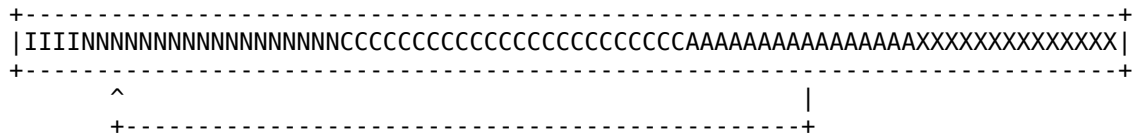
There are a myriad of technical weapons in the toolbox of the cracker most common of these are the buffer overflow, integer overflow, and heap overflow. So to know our enemy to a satisfactory point where we can understand what they can do we must first become the enemy and use their tools.

### *The Stack-based Buffer Overflow*

The buffer overflow or stack-based buffer overflow, the most common and widely known method for crackers to compromise your system. A buffer overflow is most simply the act of stuffing more data (information) into a buffer (section of memory) than it is able to hold. By doing this the attacker is able to fill up the finite size of the buffer and then begin to spill over and overwrite other variables. This is caused by how computer systems based of the Von Neumann architecture are designed. The most basic of this behavior is the property that there is no distinctions in memory of whether instructions or data is being stored. (7) Stack space is allocated down towards lower numbered memory addresses and variables on the stack grow up towards higher numbered memory. What this leads to is if a buffer in memory is overflowed with more data than what it can store. Certain functions or poor programming practices can cause the overflowed data to overwrite other variables on the stack and even overwrite the return address for the function. This is significant because the return address is the value of where to start executing instructions once the function has



NOP instruction to pad the beginning of the injected code and padding the end with the return address we can vastly increase the chances of overwriting the return address and landing in the shellcode. (3)



With this setup we don't have to be exact in knowing where the return address is located or know the exact location of where we need to jump to to being executing since there is a nice big pool of NOP instructions that we can jump into and it will eventually after a few microseconds reach our code. (3)

Now with the basic knowledge of how a buffer overflow works we'll see a few examples of what programming errors allow this behavior and some things that can be done to prevent them.

```

vulnerable.c
int main(int argc, char *argv[])
{
    char buffer[500];
    if(argc>=2) strcpy(buffer, argv[1]);
    return 0;
}
  
```

The example above illustrates a common problem with the strcpy (string copy) function. Strcpy does not perform any sort of bounds checking. Since we have fixed size string buffer that can only contain 499 character plus the null character ('\0') we can pass an argument on the command line that is greater than 500 characters thereby overflowing the buffer. By using a program like udp\_network\_exploit.c where we can specify an offset for the return address value, the operating system, which in this case is FreeBSD 6.x, and the size of the buffer that we can to send the program we can create a buffer with the form NNNNNNNNCCCCCAAAAAAAA as in the previous example. We execute the program with the the output of the udp\_network\_exploit program which is a buffer of length 700 with shellcode for FreeBSD with an offset of 100 to the return address.

```

> ./vulnerable `./udp_network_exploit 100 1 700`
shell code length: 52
Stack pointer: 0xbfbfe818
    Offset: 0x64
    Return addr: 0xbfbfe7b4
exploit code length: 699
$ exit
'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì
'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì
'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì
'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì'çìì
'çìì'çìì'çìì'çìì'çì: File name too long
  
```



check to see the length that was given is too big for the buffer. If not then we use that to bounds check the length and then use that as the max length to copy since we just check it against the buffer size. With this last assertion our supposedly safe version of string copy breaks. Since we are converting the length argument at [1] from a long with `strtol` that returns a signed short it is implicitly type casting the value. So if we enter in a large value such as `0xFFFF` it will overflow the short since it can't hold a number that large and make the number negative since it will rollover. Now with a negative length we can pass the check at [2] and when it comes to [3] to copy the string since `strncpy` requires a `size_t`, which is an unsigned integer it will assign it the negative length, but since an unsigned value can not be negative the number will be interpreted as a very large number so the length checked buffer length is now greater than the buffer size and then can be overflowed by data in `argv[1]`. This error is caused by signedness errors at [1] and [3]. (9)

The above example illustrates the subtle problems with integer overflow errors. They are hard to detect and reason about and they are also difficult to exploit since generally when a signed value rolls over it will become a very large unsigned number causing, in this case to cause up to 4GB of data to be copied or allocated assuming a 32bit platform and the program will probably cause a segmentation fault in `strncpy` and not even get to our shellcode in the buffer. Preventing bugs with integer overflows requires careful input checking and not assuming things about your input especially if you are using the input for length or size values.

### ***The Heap-based Buffer Overflow***

Heap based overflows are similar to stack-based buffer overflows described earlier, except they occur on the program heap where dynamic memory is allocated via `malloc()` and the `new` operator in C++. Heap overflows can be used in conjunction with stack-based buffer overflows to disable stack protection that uses canary values on the heap. Even if a heap is marked executable you might not need to inject shellcode in a heap-based buffer overflow since you can still damage data and cause a program to crash or cause unpredictable behavior that can be used in conjunctions with integer overflows to cause unintended behavior. These vulnerabilities like stack-based buffer overflows can be caused by insufficient checking of user input. (10)

### **Know yourself...**

There are many things that can be done to secure services that run exposed to the world. There are techniques to prevent and detect stack-based overflows using technologies

such as StackGuard and doing things such as making certain parts of the heap non-executable. Many of these techniques have a impact on performance, but most importantly are not widely deployed. (10) So starting from the ground up once we have a service compiled and ready to deploy how do different methods of running a services help secure that service from compromising the system.

### ***Running as root***

The simplest way to run a service or daemon is to just start it as the root or “super-user” in your \*nix environment. This will allow the service to access all of its files and have full access to any devices that it may need to communicate with the outside world. The downside to this is that if an attacker can compromise the service then he will have full access to the systems resources allow them to delete files, change configurations, affect other processes, and cause general mishap. In other words it's no longer your system. So therefore the simplest way to run a service is also the most dangerous since you are just relying on there to be no errors or misconfiguration in the running software.

### ***Running as an Unprivileged User***

Running a service as an unprivileged user can provide significantly more security. What a service that uses an unprivileged user does is to start as root bind to a port then shed it's privileges to the specified user so that it only has access to it's configuration and data files. When a process is running under an unprivileged user is can be constrained by CPU usage, disk quotas, and it doesn't have access to all the configuration and kernel level controls that root does. In this configuration the service cannot affect other processes running on the system if it is compromised. All services if possible should be run as an unprivileged user. To gain root access you would need a kernel exploit to elevate the process' privilege level or exploit a server misconfiguration such as storing the root password in a plain text files to which the unprivileged user has access or exploit bad policy such as passing a password as an argument on the command line that can be viewed with the ps utility to view processes. Barring the use of a virtual machine like JVM or .NET this is the highest level of access restriction that can be placed on a service running on a Microsoft Windows NT based operating system, though the use of ACLs to restrict access to some areas of the filesystem.

### ***Running in a chroot***

The next type of setup is called a chroot setup where you isolate a process to minimized view of the files system normally with its only libraries and binaries. This limits the resources that the attacker has access to if they manage to exploit a vulnerability in the

service that is running in a chrooted environment. This environment is still in the same userspace so it can still view all the other processes in the process list. This isolation is provided by the chroot() system call which first appeared in V7 Unix in 1979 (5). This provides the same protection as running as an unprivileged user, but limits the access to filesystem resources, which has the benefit of not allowing the process to see other's files. A major caveat to this setup is that depending on your setup or the service that you would like to run it can be difficult or time consuming to setup since all the necessary libraries must be copied to the chroot. This method is not any better if running as root since the chroot can be easily circumvented by calling fchdir(), then calling chdir("..") many times, then chrooting to the real root with chroot("."). Some operating systems such as the BSDs not vulnerable to this attack since they disallow chroot() if a process has any open file descriptors on an directory. (17)

## **Running in a Jail**

Jails in FreeBSD were first included in 4.0-RELEASE by Poul-Henning Kamp and Robert N. M. Watson. Its major goals are to provide a partitioned environment that will keep the standard Unix user and permissions model while allowing processes to be run in confined environments. It endeavored to go beyond the scope of just a chroot that does not have adequate protections for processes that are running as root or that gain root access. (2:1-2) For the most part jails act just like your standard FreeBSD environment with all the Unix tools and the ability to be managed separate of the host system using tools like cvsup and portupgrade to manage all the software in the jail independently. Since these jails use the same standard Unix management and user model it is able to leverage the existing knowledge of administrator without having the extra complexity of security models such as User Mode Linux or creating application level management that just increases the complexity of the code causing vulnerabilities that can sometimes be difficult to reason about the correctness. As has been shown with integer overflows since they can in many cases cause highly non-deterministic behavior. (2:2-3)

With this partitioned scheme jails allow you to have root in a jail that only affects that jail and can not view or access things outside the jail. One of the main advantages of this is that if you have to run a process as root such as part of sendmail or Apache using the RTR Frontpage Extensions from Microsoft you can do so without having to worry about it getting hacked and someone gaining root access since they will only have access to the jail. This behavior should be able to be detected and then just shutdown the jail from the host

environment and quickly(13) recreate a jail with the up to date patched software.

Securelevels are another layer to the protection scheme when using jails. It does not require a jail to be used, but is most useful in a jailed environment. There are 5 different securelevels. The first and lowest level is -1 Permanently insecure mode. This is the lowest level and no securelevel restrictions are in effect. Level 0 is the default securelevel where the restrictions to chflags(1) are observed, but can be added and removed by root. Level 1 is secure mode where the “system immutable and system append-only flags may not be turned off.” These are the flags set by chflags(1). The “disks for mounted file systems, /dev/mem, /dev/kmem and /dev/io ... may not be opened for writing,” and “kernel modules may not be loaded or unloaded.” This will prevent an attacker with root privileges from writing directly to memory, kernel memory, or I/O devices if those are exposed in the current environment. With the system immutable flag and not being able to load kernel modules is important since even if an attacker gains root access they will not be able to trojan any binaries with the system immutable flag or install a kernel rootkit by loading a kernel module. Level 2 is highly secure mode, which is the “same as secure mode, plus disks may not be opened for writing (except by mount(2)) whether mounted or not. This level precludes tampering with file systems by unmounting them, but also inhibits running newfs(8) while the system is multi-user. In addition, kernel time changes are restricted to less than or equal to one second.” Finally Level 3 the highest securelevel is network secure mode. This is the “same as highly secure mode, plus IP packet filter rules (see ipfw(8), ipfirewall(4) and pfctl(8)) cannot be changed and dumynet(4) or pf(4) configuration cannot be adjusted.” Where this can come in handy in protecting against break-ins is the firewall on the system is only configured to allow incoming http connections on port 80 and all other inbound and outbound connections are denied. With this setup since the firewall rules can not be changed a common technique for downloading additional shellcode that doesn't fit in the buffer from a buffer overflow is to connect to a remote server and download and execute the additional code can be prevented forcing the attacker to use another method to download additional code like the metasploit code `bsd_ia32_findrecv_stg` (15) that only works if your program has an open file descriptor. What makes the whole securelevel system work is that you can only increase securelevels while in multiuser mode. (14)

Even with the protection of a jail and securelevels there is still the possibility of an attacker gaining root access in the host environment since the jails and the host environment share the same kernel (12). Using our knowledge of exploits lets see how this was done in “Exploiting Kernel Buffer Overflows FreeBSD Style”.

The first step was to find a vulnerability that any user would be able to execute whether jailed or not. This manifested itself as a common stack based buffer overflow in the jail utility not correctly bounds checking the length of the hostname. Now this only gets us access to run commands as the current user, so we need to find a kernel vulnerability. This was found in how the jail command used procfs to handle process information with the pbuf variable with the use of the sprintf() function for formatting strings. You might remember the sprintf() function from our previous list of dangerous functions. Using a GENERIC kernel and by adjusting the optimization levels while using a disassembler to see how the compiler generated the code we can determine that the same code is generated in debugging and -O optimized mode so we can use one exploit to handle all cases on the target architecture. Now that we figured out that we can get kernel access we must figure out a way to break out of the jail. When in kernel mode we can access kernel memory via kldsymb(2) and kvm(3) it requires simply changing address values on the prison structure that hold the jail information by calculating the offset of the address. Now that we can break out of the jail the need to exit gracefully is a must, since unlike a user process that can bomb out and we don't care, if a kernel process does not exit correctly it can cause undefined behavior in the system or worse cause a kernel panic thus halting the machine. Since at our current execution location we can not calculate the correct locations for the return address and frame pointers we will just sleep() the process and come back in with a second attack and clean house since we will then be able to calculate all the necessary addresses allowing the process to exit out of kernel mode without crashing. The way we can intercept when the process that we called sleep on is to "hijack" the entire system call handler to call our code to fix up the sleeping process and then call the real system call handler to finish up the job. Since in this example the host environment has a securelevel of -1 the firewall can easily be modified to change it's rules. Since we have root in the host environment with securelevel -1 we have full control and can recompile the kernel and install a kernel rootkit. (16)

The above example shows nothing is completely secure and an attacker with enough time and knowledge of the system will be able to find a way to break it, but an attack of this level of depth requires an attacker to be highly knowledgeable about how the operating system works and there is only so much that can be done against an attacker with this level of skill. Since the article was written the vulnerability in procfs and jail was fixed and due to the fact that procfs on FreeBSD was inherently flawed (16). Procfs has been removed from being mounted by default on the current production release 5.3 and later.

### **You need not fear...**

With the different methods of trying to secure services from unprivileged user to jails with restricted firewalls and securelevels the best approach would not just employ a single method, but would layer the protections. This way it will force an attacker to spend more time trying to break into the system thereby giving an IDS system a better chance of detecting the malicious activity. Even with all the attack vectors there is still a lot that can be done to protect a service from outside attack. “Thus the highest form of generalship is to balk the enemy's plans, the next best is to prevent the junction of the enemy's forces, the next in order is to attack the enemy's army in the field, and the worst policy of all is to besiege walled cities.” (11) So listen to what Sun Tzu said and force them to attack your “walled city” or a server.

## Bibliography

- 1) <http://lists.freebsd.org/pipermail/freebsd-current/2004-March/024819.html>
- 2) <http://phk.freebsd.dk/pubs/sane2000-jail.pdf>
- 3) <http://www.phrack.org/phrack/60/p60-0x06.txt>
- 4) [http://www.sun.com/bigadmin/features/articles/solaris\\_zones.html](http://www.sun.com/bigadmin/features/articles/solaris_zones.html)
- 5) <http://www.faqs.org/faqs/unix-faq/faq/part6/section-2.html>
- 6) <http://www.collusion.org/Article.cfm?ID=176>
- 7) <http://www.csupomona.edu/~hnriley/www/VonN.html>
- 8) [http://www.freebsd.org/doc/en\\_US.ISO8859-1/books/developers-handbook/secure-bufferov.html](http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/secure-bufferov.html)
- 9) Phrack #60 Article #0x0a
- 10) <http://www.w00w00.org/files/articles/heaptut.txt>
- 11) <http://www.military-quotes.com/Sun-Tzu.htm>
- 12) FreeBSD 6-CURRENT jail man page
- 13) make-jail.sh
- 14) FreeBSD 6-CURRENT securelevels man page
- 15) [http://www.metasploit.com/shellcode\\_bsd.html](http://www.metasploit.com/shellcode_bsd.html)
- 16) <http://cert.uni-stuttgart.de/archive/bugtraq/2000/12/msg00500.html>
- 17) <http://www.bpfh.net/simes/computing/chroot-break.html>
- 18) <http://am-productions.biz/docs/security-project.tgz> (This contains all the code samples and references to external files plus more stuff that was not in the paper.)