Mitigating Buffer Overflows by Operating System Randomization[1]

Monica Chew   Dawn Song

December 2002

CMU-CS-02-197

Abstract

We propose three methods for mitigating buffer overflows by using operating system randomization: randomization of system call mappings, randomization of global library entry points, and randomization of stack placement. These mechanisms are light weight. They increase the heterogeneity of systems and make it more difficult for attackers to exploit buffer overflow vunerabilities. We also present a mechanism, *timed capabilities*, for more secure resource management.

[1]Monica Chew: mmc@cs.berkeley.edu, UC Berkeley; Dawn Song: dawnsong@cmu.edu, Carnegie Mellon University. This work was mainly done in Fall 1999 at UC Berkeley.

Keywords: Buffer overflow, software security

1 Mitigating Buffer Overflows

1.1 Buffer Overflow Review

The most common buffer overflow occurs when an attacker writes past the end of an array and onto the return address of the stack frame, as illustrated in Figure 1. A malicious buffer overflow exploit will typically fill the array with assembly code that executes a shell and overwrite the return address with the address of the beginning of the array. Thus, a shell is spawned during execution. If the program happens to be setuid root, the spawned shell will be a root shell, and the attacker gains control of the victim machine. Buffer overflows are the largest single class of exploits [2].

Typically, the assembly code, or shellcode, sets up some arguments and calls `execve("/bin/sh", ...)`. The way to make a system call is generally to push the index of that system call onto the stack and force an interrupt. In Figure 1, `0xb` is the index of execve, and `int $0x80` is the interrupt instruction.

Another type of exploit uses the C library, `libc` . The return value on the stack frame is overwritten with the address of a function (e.g. `system()`) in the library [8]. Library exploits are typically used on non-executable stack architectures.

1.1.1 Necessary Conditions

In order for a buffer overflow attack to successfully interact with the operating system, either condition 1 or 2 must be satisfied, and for stack-based buffer overflow attacks to succeed, condition 3 must be satisfied:

1. Known system call mappings. The buffer overflow exploits that make system calls directly must know the index of the system call to push on the stack.
2. Known library entry points. Exploits which use the library must know the address of the function to call.
3. Known stack placement. All exploits must know where the stack begins, since the return address is absolute.

In a 1999 survey of three months of published buffer overflow exploits, approximately 85% of published exploits use the system call approach, and the other 15% are library exploits [3]. We investigate ways to make all three of these conditions more difficult to satisfy.

1.2 Defenses against buffer overflows

The approaches chosen were based on a few observations. Current tools for finding bugs are not good enough. According to one study, 40% of all applications halt on random input [6]. So, buffer overflows will always happen, but we can try to limit the amount of damage done by remote attacks.

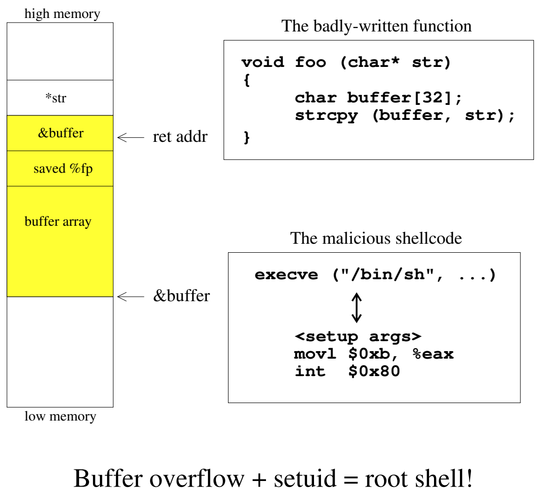

Figure 1: Anatomy of Buffer Overflow

A different approach can be used to solve each of the three necessary conditions for successful exploits. These approaches were chosen with the goal of increasing the work factor for successful exploits and increasing the heterogeneity of systems [1].

1. Randomizing system call mappings. System calls have well-known indices into the call table. Historically, these are kept in /usr/include/asm/unistd.h. By obfuscating their mappings, an attacker will have a much lower probability of knowing the correct index.

2. Changing library entry points. Since library exploits must know the address of library routines, we can try to obfuscate these as well.

3. Randomizing stack starting point. By randomly placing the stack, the attacker will guess the correct return address with much lower probability.

A combination of these approaches will increase the work factor needed by traditional attacks. Since these approaches either have built-in entropy or are easily configurable, they also increase the heterogeneity of systems, making vanilla exploits much more difficult to write.

Besides these common cases, there are some attacks which do not use system calls. For example, in 1994 there was a bug in sendmail that allowed the uid to be overwritten to 0, the uid for root [13]. Also, some buffer overflow exploits use the heap instead of the stack [10]. Our approaches do not cover these attacks. However, these attacks are rare and did not show up in our investigation of three months of exploits.

1.3 Method 1: Randomizing system call mappings

All exploits that directly make system calls use hard-coded system call numbers. These numbers are indices into the table of system call function pointers. Obviously, if the number is incorrect, the correct system call will not be made. The solution is to change the system call mappings. First, we recompiled the kernel with different system call mappings. Then, we rewrite system call numbers in statically linked legitimate code, either on the fly at load time or between compilation and execution as a preprocessing step. In addition, we double the size of the system call table to 512 to increase the search space. Legitimate code will be rewritten correctly, but code injected at runtime will not be rewritten as shown in Figure 2.

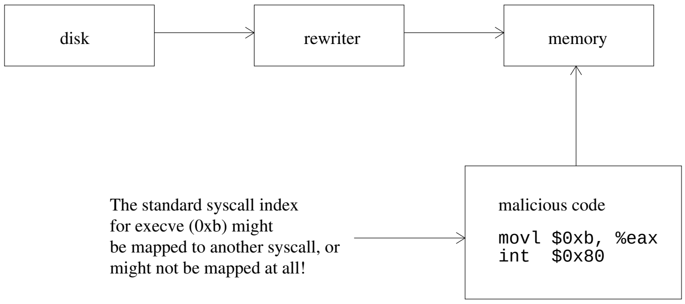

Figure 2: Binary Rewriting

Both load time rewriting and preprocessing incur very little overhead. The load time rewriter is invoked when a page is read off the disk and adds 0.1% to the cost of I/O. However, once an executable is read off of the disk, it is likely to stay in memory, so it does not have to be rewritten every time it is run. The preprocessing approach has the advantage that it only needs to be done once. On an IBM Thinkpad 560Z (300 MHz Pentium II processor with 64MB RAM), the execution time is 0.7 ms/KB. Approximately 90% of this cost is due to disk traffic. On a default RedHat 6.0 installation, fewer than 15 binaries (including libraries) need to be rewritten, the largest of which is 19MB (`/usr/lib/libc.a`).

Although randomizing system call mappings has a high probability of making the attack fail, a buffer overflow could still lead to a core dump. This is acceptable, since all we are trying to prevent are root exploits, not buggy code.

There are ways of defeating this method. If this system were widely adopted, future generations of buffer overflow exploits would be forced to scan through the currently running executables for the true system call mappings before making the call.

1.4 Method 2: Changing Library Entry Points

From our investigation, 15% of the exploits to gain root shell use functions in dynamically linked libraries [11], e.g. the C library `libc` , instead of using hard-coded system call numbers. In order to do this, the malicious code has to know the exact address of the desired `libc` function entry points. In the current operating system design and implementation, the shared libraries are usually mapped into the same virtual address on similar systems. So the malicious code can have a good chance of guessing of the `libc` function entry points on a remote host. Hence, if we can change the `libc` function entry points on different hosts, the malicious code can no longer guess the `libc` function entry points and will not succeed.

There are many ways to change the `libc` function entry points. One way is to change the original functions to dummy functions and insert new functions which provide the same functionality as the original functions. The dummy functions can add auditing capabilities, such as notifying an intrusion detection system that the dummy function has been called. As illustrated in Figure 3, we change the function `execve` in glibc into a dummy function, and we insert a new function `safe_execve` which does exactly the same thing as the original `execve`.

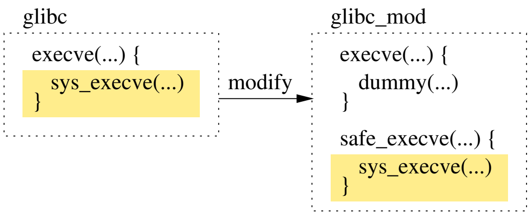

Figure 3: Changing Entry Points in glibc

After changing the `libc` function entry points, the normal program will still be compiled using the original entry points. In order to make the legitimate program to work, we can perform a simple binary rewriting. During the binary rewriting, the rewriter will simply search and replace the original function name (i.e. `execve`) with the new function name (i.e. `safe_execve`) in the `.dynstr` section in the binary. The `.dynstr` section is the collection of strings of the dynamic linking symbol table. In order to keep the rewriting simple, the length of the new function name should be the same as the original function name. As illustrated in Figure 4, because the injected malicious code will not get rewritten, it will have the wrong entry points and hence will not be successful.

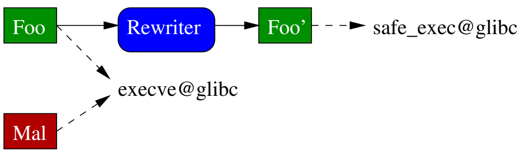

Figure 4: Binary Rewriting for Library Entry Points

We can perform the binary rewriting either during dynamic linking time or before loading. To do the binary rewriting before loading, we only need to do it once so there is no runtime overhead. But since the user would need to perform the rewriting, it is not transparent to the user. To do the binary rewriting during dynamic linking time, we need to change the dynamic linker and loader. Whenever the dynamic linker and loader tries to resolve the original function symbol (i.e. `execve`), it will return the resolution for the new function symbol (i.e. `safe_execve`) (This is not implemented). Because the rewriting is simple, it causes little runtime overhead, although it needs to be performed every time when the program is executed. Since the user does not need to perform any command for the rewriting, this way of rewriting is transparent to users. The total time overhead of the rewriting here is even cheaper than the rewriting in the case of randomizing system call mappings, because the rewriting is only performed on the `.dynstr` section instead of the whole `.text` section.

Potentially there are ways to get around this method. The malicious code could walk up and down the stack and try to find the library entry points used by the victim program or it could scan through the library pages in memory to try to find out the right entry points. However, this certainly requires much more work than is found in a typical exploit.

1.5 Method 3: Randomizing Stack Placement

The stack-based exploits need to know the address where the injected malicious code resides on the stack and put it as the return address. In Linux 2.2.x, the stack of every process starts at exactly the same address (e.g. `xc0000000`). Hence the same program running on similar systems would have very similar stack placements, and the malicious shellcode can guess the right return address before hand. If we let the stack start at a random position for each process, the malicious code can no longer predict the right return address and hence cannot start running.

Upon initializing the process, the loader puts certain data on the top of the stack and then sets the stack pointer and starts the process. We modify the loader so that before it copies the environment variables and argument variables onto the stack, it generates a random number and pushes the stack down by this random value, as shown in Figure 5. Because of the implementation details, the random value is limited to the maximum number of argument pages multiplied by the page size specified in the system. Up to Linux 2.2.x this number is about 32K.

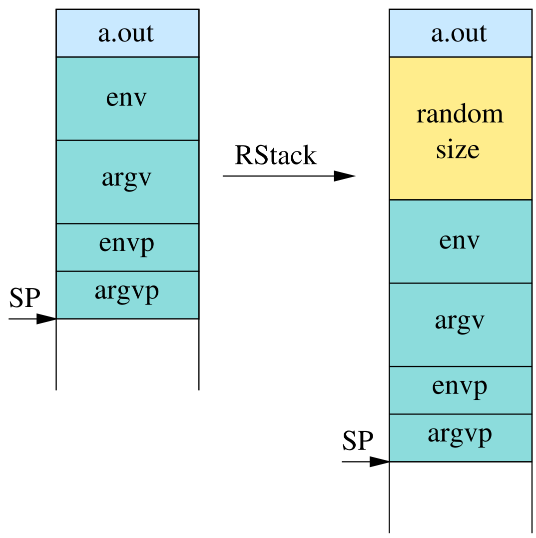

Figure 5: Randomizing the Stack Placement

Moving the stack introduces almost no runtime overhead although it wastes a certain amount of memory. Because we do not need to actually allocate physical pages for the empty pages we introduced, the maximum amount of physical memory wasted is two pages. A more complicated implementation could result in a scheme where the maximum amount of memory wasted is only one page.

Malicious code actually does not need to know the exact return address. It can insert a sequence of NOPs in front of the real code so that it can jump anywhere into the sequence of NOPs and then continue until the real code starts. In our method, since the randomization value is about 16K on average, the malicious code would need to include a long sequence of NOPs to break this method, which is not always realistic.

2 Timed Capabilities

For many resources, the current systems usually only check the permissions at the beginning when a process request a resource. And once the permission checking is done, the process will always have the capability to use the resource. This design could lead to security holes in many scenarios. For example, the graphics library `sgvalib` opens `/dev/mem` as root and then drops root privilege after initialization [4]. However, the file descriptor for `/dev/mem` remains open. An attacker exploiting `svgalib` does not directly get a root shell, but gains read/write permission over all memory, and so can overwrite the uid with 0. Hence, we would like to provide a method to limit capabilities for system resource usage.

One way is to provide a timed capability. Namely, each capability token has a timeout field associated with it. When using the timed capability to request resource usage, the system checks whether the capability has already expired. If capability has expired, the system tries to re-verify the permission. If the re-verification fails, the system returns an error.

For a proof of concept, we implemented a timed file descriptor. We add a new system call to the kernel, `sys_topen(char * fn, int flag, int mode, unsigned long timeout)`. We also add the timeout field into each entry in the file descriptor table of each process. If the process wishes to open a timed file descriptor, it uses the `sys_topen` system call to specify the timeout and set a particular bit in the `flag` to specify that the timeout field now is valid. Upon each read and write, the system will check whether the particular bit in `flag` is set. If set, it will then check whether the file descriptor has already timed out. If it is, it will try to recheck the permission. If the permission cannot be verified, it will return error. In this way, we can provide a simple timeout policy and retain backward compatibility. Using the `sys_topen`, a process can either specify a limited timeout value or specify that it should never timeout. For a process which opens a file descriptor using the normal `sys_open` system call, the result will be the same as specifying an infinite timeout value, and hence programs which do not have the notion of `sys_topen` will not be affected at all. This method also causes little runtime overhead since it will only need to check a bit in the flag and compare the timeout value with the current time.

3 Related Work

Other projects that have tried to stop buffer overflows include StackGuard [5] and Stack Shield [12]. StackGuard injects code at compile time into the prolog to insert canaries (magic bytes) near the return address of the function. It also puts the original canaries in the data segment. If the canary changes, then the program halts. Stack Shield is similar, but stores the return address instead of the canaries. The runtime overhead of StackGuard is 30%.

Richard Jones and Paul Kelly [9] have implemented C bounds checking at compile time. This approach results in a 10x slowdown at execution time.

Stephanie Forrest et al. have proposed of changing the stack placement [7]. Their approach was to modify `gcc` to pad the stack with random number of bytes in front of each function at compile time. Thus the same source code would compile into different binaries with different stack placement. But this approach potentially wastes much more memory space than our approach and the same binary will always have the same stack placement.

4 Future Work

Currently we are using nine bits for the system call index, so the search space of the attacker is only 512 system call numbers to try. If we increase the size of the system call index to one word, then the attacker would have very little chance of even getting a valid system call, since there are only 190 system calls in Linux 2.2.x. Because the system call table has to be continuous in the current kernel, increasing the size of the index to one word would require a 16GB system call table. A more efficient implementation would require changing the way the system call table is dereferenced. Furthermore, by increasing the size of a system call, we can ensure that all of them contain null bytes. Since many buffer overflows occur during string functions (e.g. `strcpy()`, `strcat()`), this would make exploits even more difficult.

Current implementation of rewriting symbols in the `.dynstr` section is done before loading time. Another way of rewriting is to modify the dynamic linker and loader to do the rewriting on the fly.

Although we did development on Linux, the ideas are portable to other operating systems. We only chose Linux because it is freely available and easy to modify. It would be very useful to apply these techniques to Windows and Solaris because they are also popular targets. In the case of Windows, it would be more difficult to re-map system calls without the kernel source, but still possible.

Also, a more intelligent timeout policy for file descriptors is needed. Currently timeout values for file descriptors are determined statically with the call to `sys_topen()`. A bad timeout value could either lead to poor execution time, if the timeout is too short and permissions must be rechecked often, or ineffectiveness, if the timeout is too long. An adaptive timeout policy might be more useful. Also, it would be good to implement action-based policies. For example, it would be just as easy to implement a policy that is invoked whenever a dangerous system call is made (e.g., `setuid()` and `setgid()`) to be used in addition to the timeout policy.

5 Conclusion

We can increase the security of Linux by applying a few small patches that make buffer overflow exploits more difficult, and operating system resource leaks less devastating on average. The cost of these approaches is negligible (0.1% overhead of disk I/O, in case of dynamically rewriting system call indices). Although all of the methods can be circumvented, it would take some time before these changes were propagated through the exploit world, and it is likely that many existing exploits would be made obsolete because of shellcode constraints.

Despite all of this, we feel that these approaches are lightweight and effective enough to deserve consideration when trying to improve security.

Acknowledgements

We would like to thank George Necula for the suggestion of randomizing stack placement. We would also like to thank David Wagner for the sendmail exploit example and Adrian Perrig for figuring out how to make new system calls.

References

[1] *The Protection of Information in Computer Systems*, chapter Basic Principles of Information Protection. http://www.mediacity.com/~norm/CapTheory/ProtInf/, 1975.

[2] Attack class: Buffer overflows. http://helloworld.cs.concordia.ca/1999/04-apr/attack_class.html, April 1999.

[3] Various authors. rootshell. http://www.rootshell.com, Feb and Jun 1999, Dec 1998.

[4] bugtraq. An svgalib vulnerability. http://www.arava.co.il/matan/svgalib/hypermail/0117.html, 1998.

[5] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *In 7th USENIX Security Conference*, 1998.

[6] Barton Miller et al. Fuzz revisited — a re-examination of the reliability of unix utilities and services. http://www.cs.wisc.edu/Dienst/UI/2.0/Describe/ncstrl.uwmadison/CS-TR-95-1268, April 1995.

[7] Stephanie Forrest, Anil Somayaji, and David Ackley. Building diverse computer systems. In *In HotOS-VI*, 1997.

[8] Horizon. Non-executable stack exploit. http://www.securityfocus.com, March 1999.

[9] Richard Jones and Paul Kelly. C bounds checking. http://www-ala.doc.ic.au.uk/~phjk/BoundsChecking.html.

[10] Jeremy Kothe. An exploitable heap overflow in windows 95/98 telnet. http://www.securiteam.com/exploits/An_exploitable_Heap_Overflow_in_Windows_95_98_Telnet_exe.html, December 1999.

[11] Nergal. Defeating Solar Designer's non-executable stack patch. http://www2.merton.ox.ac.uk/~security/bugtraq-199801/0152.html, 1998.

[12] Vendicator. Stack shield. http://www.angelfire.com/sk/stackshield.

[13] Zhart. Collection of sendmail exploits. http://bau2.uibk.ac.at/matic/buglist.htm, November 1996.