Guarded Memory Move (GMM)
Buffer Overflow Detection And Analysis

Davide Libenzi
davidel@xmailserver.org

January 17, 2004

Abstract

Buffer overflow attacks have become more and more frequent with the advent of the Internet. The public exposure of network servers and the increased popularity of the Internet, have increased dramatically attempts to hijack publicly available servers. The advent and the increased popularity of Open Source also, while increasing the stability and the architecture of existing software through cooperation, gave hackers the key to spot vulnerabilities and to exactly calculate frame addresses. Many famous packages like bind, wu_ftp, apache and sendmail, just to keep the list short, have been subject of buffer overflow vulnerabilities. When running a public Internet servers, it is always a good idea to harden your system using tools like stack execution protection, random binaries relocation and so on. Those methods, while giving protection about remote exploits, are very poor when it comes to forensic analysis and debugging of the attack. It is the purpose of this document to introduce a new method called *Guarded Memory Move* or, more shortly, **GMM**.

Introduction

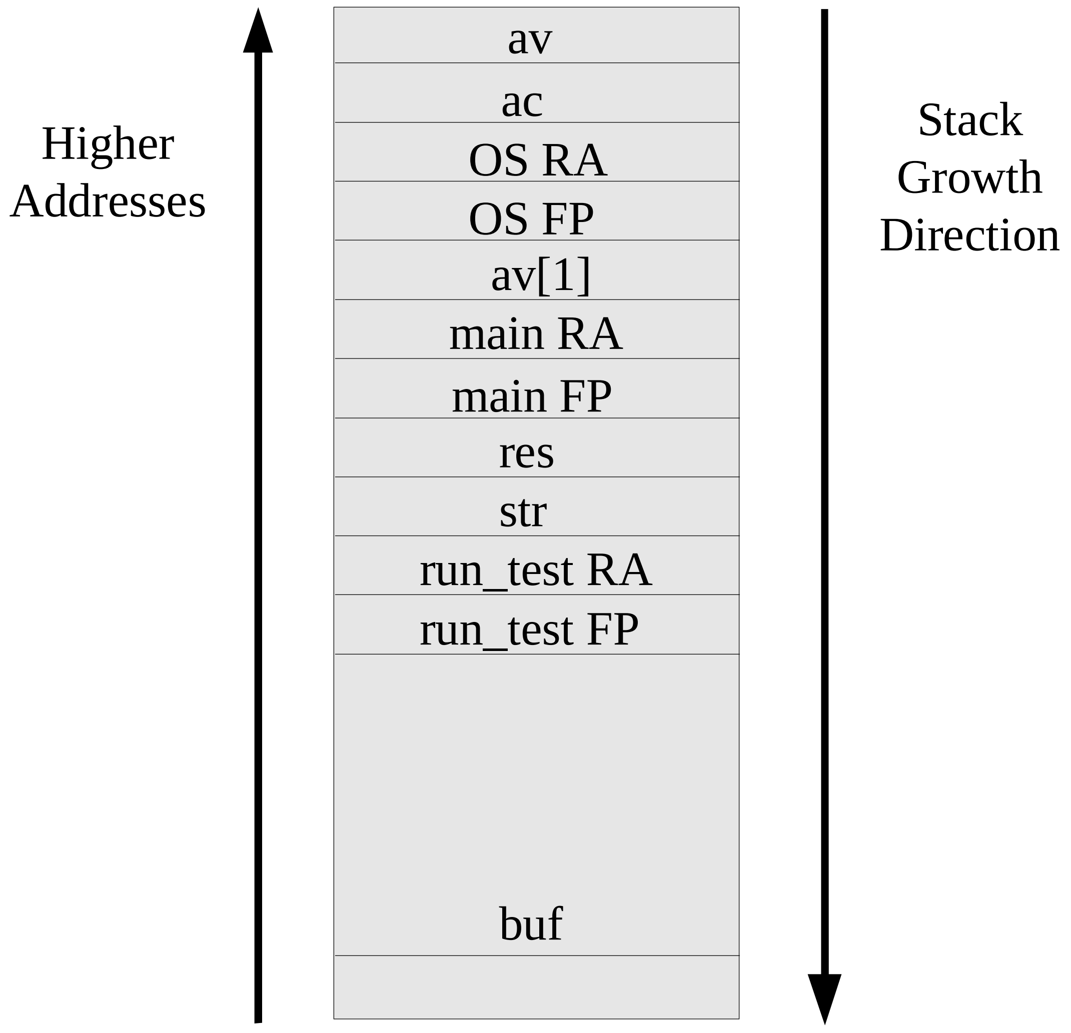
An increasing number of buffer overflow exploits is reported year after year with huge damages reported by companies hit by hackers taking advantage of such vulnerabilities. The result of a successful buffer overflow exploit is very serious, since it allows the attacker to run its own code on the victim machine. Using this back-door, the attacker can inject its own binaries and libraries and, if the attacker has been careful, no signs are given to the administrator about the exploit itself. The machine will continue to run normally, with an exception. Malicious code will be sitting on the exploited machine waiting for the attacker to trigger it. A number of tools are available to limit or avoid damages related with a buffer overflow vulnerability. Those ranges from local and network intrusion detection systems, up to stack protection user-space, kernel patches, gcc patches and user-space libraries. While those tools can be very good in stopping or limiting the attack, they do nothing in helping the administrator to solve the software problem and/or to report a thorough bug report to the group responsible of the server development. The real problem is that, when the anomaly is caught, the application stack frame has already been permanently damaged and no information are available for a forensic analysis of the problem. It is of vital importance to give to developers both a valid stack trace and a good dump of functions parameters and local variables, since they reveal two crucial information. First, they spot where the problem is in the source code by allowing them to concentrate their focus on a limited portion of the whole listing. When dealing with applications made by hundreds of thousands of lines of code, this is always an appreciated hint by developers. Second, it reveals the data used by the attacker to exploit the source. Many times, even when a precise location in the source code is spotted, the data/conditions that make the algorithm to fail might not be clear. This especially when the algorithm is not trivial and has many data dependencies. It is in this scenario that **GMM** comes into play, by giving precious information to study and resolve/report the vulnerability.

Anatomy Of A Buffer Overflow Exploit

A buffer overflow exploit born from a programming error in the target application source code. Languages like, for example, C are more subject to those kind of errors compared to more high level languages like Java, Perl, Tcl/Tk, Python, BASIC, etc... Even more experienced C programmers are not free from committing such errors, especially when dealing with the development of huge numbers of lines of code in a limited period of time. The direct access to the process memory that the standard C API allows, gives the developer both power and increased sources of errors that lead to buffer overflows. The advent and the steady growth of the Open Source community also, while giving to end users an increased and more advanced number of softwares, allows a malicious hacker to directly spot the possible source of buffer overflows by directly reading the application code. Once a C application is built, the resulting binary code is stored inside a system specific executable file, that in case of Linux (and many Unixes) is the ELF file format. The C compiler and the linker work together in building the resulting ELF file, that is composed by different sections. Without going any deeper, we can say that at least a data section (.data and .bss) and code section (.text) are present inside the ELF file. When the operation system is required to load the ELF file, it creates virtual memory regions according to values stored inside the executable file, and maps ELF file sections onto the created memory regions. One of the information contained inside an executable file, is the address of the first CPU instruction that should receive the control of the application once the OS has completed the load operation. All CPUs have a special register typically called IP (Instruction Pointer), that the operating system will set to the virtual memory address contained inside the executable file. But before giving control to the application binary code, another virtual memory segment must be created by the operating system. To keep track of the function call chain and to allow re-entrance, CPUs have (or offer the capability of) a stack register, typically called SP (Stack Pointer). Such register points to a special memory region called program stack. It is here that function local variables are allocated, and it is also here that the CPU saves the return address where requested to branch to another function, together with parameters needed by the called function itself. It is the CPU stack manipulation though over-run buffers, that gives the attacker the ability to inject and execute its own malicious code. Let's examine a simple application like:

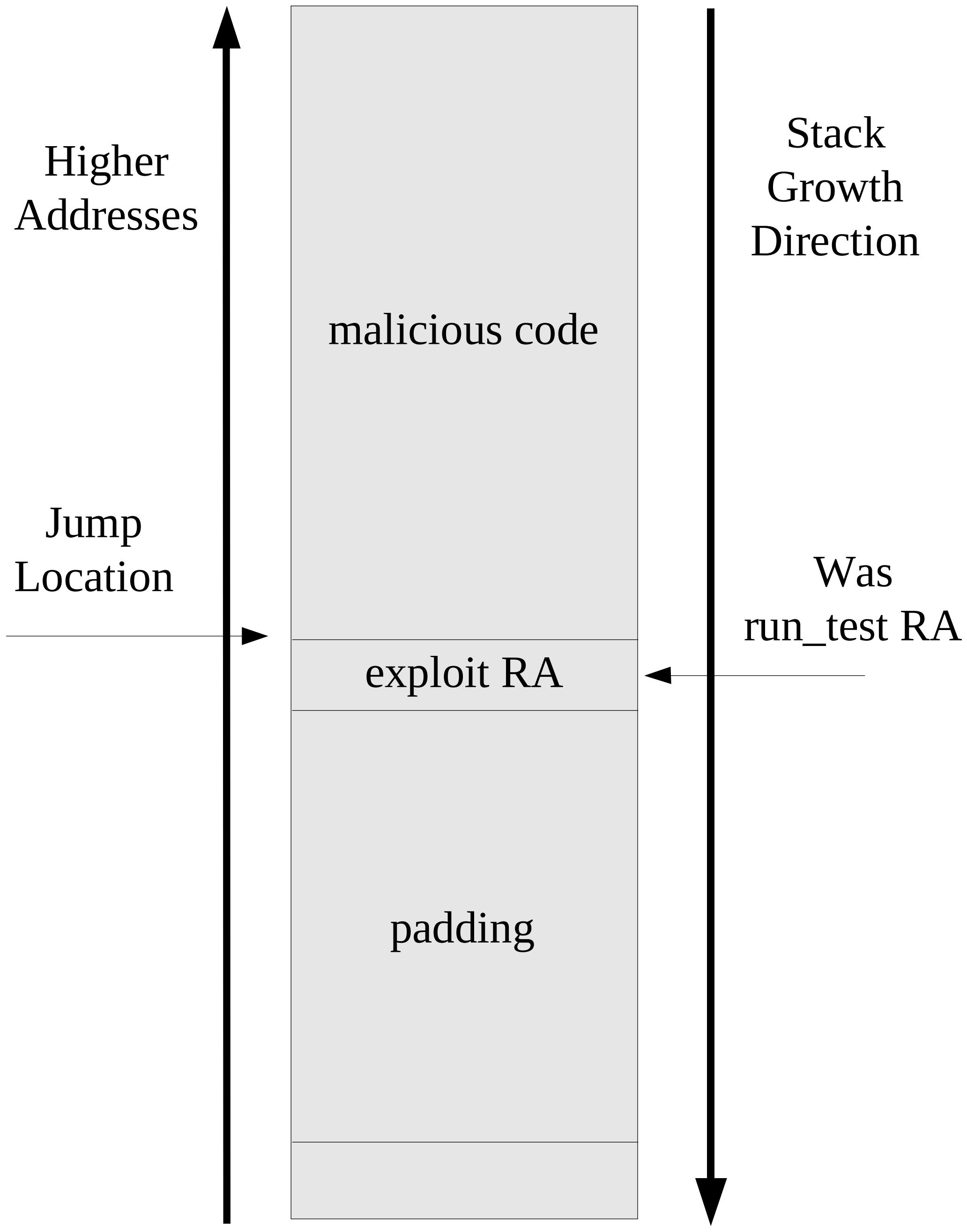
```
int do_overflow(char *str) {
    char buf[32];

    strcpy(buf, str);
    return strlen(buf);
}

void run_test(char *str) {
    int res;

    res = do_overflow(str);
}

int main(int ac, char **av) {

    run_test(av[1]);
    return 0;
}
```

This application does basically nothing but gives the user the ability to over-run the application stack by passing an over sized parameter. From an application point of view, the first instruction that will be executed will be the first instruction of the *main()* function [1]. Let's see how the stack frame looks like (supposing a stack growing towards lower addresses) right before the call to *strcpy()*:



1 This is not true in reality since other functions like constructors and library initialization functions take the control before *main()* executes.

The first two parameters that appear on the top of the stack (*ac* and *av*) are pushed in by the library/OS initialization code. When the library/OS calls *main()*, the CPU pushes on the stack the return address that points to the next CPU instruction following the call to the *main()* function (OS RA). When *main()* enters, the first thing it does is to save the current frame pointer (FP), that happens to be the library/OS one (OS FP). Having to call the *run_test()* function, that requires a string parameter, the content of the *av[1]* variable is pushed into the stack (*av[1]*). The following call to *run_test()* will make the CPU to save the address of the CPU instruction following the call to *run_test()* into the stack (main RA). The *run_test()* function will start by saving the current frame pointer into the stack (main FP), and then will allocate space for the *res* variable by subtracting the size of the *res* variable itself from the current stack pointer (SP). The next call to *do_overflow()* will require *run_test()* to push the content of the *str* variable into the stack. As a consequence of the call to *do_overflow()*, the CPU will save the pointer to the next CPU instruction following the call to *do_overflow()* into the stack (run_test RA). On entry, the *do_overflow()* function will push the current frame pointer (run_test FP) into the stack and will allocate space for the buf variable by subtracting the size of buf (32 bytes in this case) from the current stack pointer (SP). At this point the stack layout will look like the one described in the picture above. The next call to *strcpy()* is supposed to copy the content of the string *str* to the locally allocated buffer *buf*. If the length of *str* is less than the size of *buf* (32 bytes in this case), the content is copied and on exit from *do_overflow()*, the space allocated for the local variables (*buf in this case*) is deallocated by adding to the stack pointer the total size of the allocated variables (32 bytes in our example), the previous frame pointer (run_test FP) is popped from the stack and the following CPU's RET instruction will cause the CPU itself to pop the next word in the top of the stack (run_test RA) into the CPU's instruction pointer (IP). From here, everything unwind nicely and the application terminates in a clean way. But what happens if the length of *str* is greater that the size of *buf*? The following data stored after the buf array are the previous function frame pointer (run_test FP) and the previous function return address (run_test RA). By over-writing the previous function return address with random data, will make the CPU to pop a likely bad address from the stack and to jump over that address to fetch the next instruction. But what happens if the content of the over-written data is not random? What happens if someone, knowing the programming error living inside the software, try to inject carefully crafted data into the application stack? Well, as you have probably guessed, it is possible in this scenario to alter the return address to make it point to a location on the stack and to inject there malicious code into the application. But how and how difficult is to perform such a task? It is fairly simple indeed and does not require any special genius like many think. There are different techniques to do this and all lead into the calculation of the value to over-write the return address with. With one technique the calculated value will be such that will make the CPU to directly jump into the injected code. The other one will make the injected return address to point to a special instruction (jump to the address contained inside the SP register) contained in the virtual memory space of the application, that will make the CPU to indirectly jump into the injected code. After the exploit and before the CPU will execute the following RET instruction, this is how the stack frame will look like:


Higher Addresses
Stack Growth Direction
malicious code
Jump Location
Was run_test RA
exploit RA
padding

Existing Solutions Analysis

I will start by saying that many CPUs have, among the protection bits of their page table entries, the execution flag (EXEC) that allow the OS to carefully clear such flag for virtual memory regions allocated for the stack. It happens though, that design choices made by user-space libraries require the stack to be executable because they use it for things like trampolines, etc... Having an executable stack opens the door to buffer overflow exploits, and require care about application that are directly exposed to the Internet (or any non-trusted network). One class of solutions available to the system administrator are the usage of specific OS specific patches, like the exec_shield[3] patch for the Linux OS. Other solutions like StackGuard[4] and StackShield[5] work by patching GCC[6] to perform additional checks at the entry and exit from every called function. Once recompiled with the specially crafted GCC copy, applications are able to catch buffer overflows trying to over-write return addresses and are able to trigger a segmentation fault before the injected code is executed by the following CPU' sRET instruction. Both StackGuard and StackShield are available for the Intel x86 CPU family only at the current time. Another solution more similar to **GMM** is LibSafe [7]. It works like **GMM** by using the *LD_PRELOAD* environment variable to insert itself before GLIBC[8] in the dynamic linking resolution. It is then able to execute prologue and epilogue code before and after the core GLIBC function is invoked. The return address is saved into a non-exploitable memory region before the call to the core GLIBC function and it is used on the return path to verify the correctness of the one present after the core GLIBC function returns. If they differ, different actions can be performed, from sending emails, to logging with syslog, to printing warnings on stderr. LibSafe at the current stage, does not cover functions that let the user specify the byte count when copying data, and this can arise the exposure to possible integer overflows in the size calculation, that can in turn make the operation to be buffer overflow prone. Since attackers do have to guess address information from the target machine, the usage of tools like Prelink[9] can also greatly help in hardening publicly exposed machines from buffer overflow attacks. Prelink works by randomly generate mapping addresses for existing libraries and applications, so that it is almost impossible for the attacker to guess virtual memory addresses used in the target machine. While each of the above solutions enable the administrator to block buffer overflows from happening, none of them enable a forensic analysis of the exploit. And this is due a very basic reason, the content of the stack is already compromised when the anomaly is detected. This means that stack trace, function parameters and local variables are no more inspect able by the examiner. This is the main reason that drove the development of **GMM**.

GMM Implementation

During the ELF executable file loading, a special section of the ELF file is examined to extract the dynamic loader for the current file. With modern Linux-GLIBC systems this is usually */lib/ld-linux.so.2*. During the dynamic symbol resolution phase, the content of the *LD_PRELOAD* environment variable is inspected to see if any library is required to be pre-loaded before the dynamic linking begins. Using this technique **GMM** is able to insert itself before the GLIBC library loading and it is then able to replace certain particular functions with its own copy of the ones. The list of the intercepted functions include *strcpy()*, *gets()*, *fscanf()*, *sscanf()*, and many other that the reader can verify inside the **GMM** source package[10]. The **GMM** users can extend the list of intercepted functions by using the WRAP macros defined in *gmm.h* to add wrappers to *gmm.c*. It is also possible for the user, to use **GMM** wrapper macros to shield his own application functions by expanding **GMM** usage to every piece of code that has been hit by overflows. What the GMM wrapper code does is fairly simple. It saves a chunk of stack located right above the monitored function frame and the chain composed by the previous three return addresses, into a private area. Then the core function (GLIBC or application one) is called with the proper parameters. When the core function returns, the previous chain of return addresses is compared with the saved one. If they match, the intercepted function returns normally and the application will continue to run without interruptions. If the **GMM** wrapper code will instead detect a mis-matching between the saved context and the current one, an exceptional handling is performed. First of all, the previously saved stack chunk is copied back in place by hence restoring the full application stack frame. Then, if the environment variable *GMM_FAULT_EXEC* is set, the **GMM** fault handler code will execute the binary stored in such environment variable, by waiting the child process to terminate before going to execute an invalid instruction that will cause a segmentation fault followed by a core dump. The GMM core code uses GCC extension functions *__builtin_return_address()* and *__builtin_frame_address()* to retrieve the previous return addresses and frame addresses in a system independent way. This makes the solution to be GCC dependent, by this is considered a non-problem given the widespread acceptance of GCC in the Open Source community. **GMM** does not require existing applications to be re-compiled to work, even if the condition of having built the software using frame pointers is needed by **GMM**. This is usually not a problem since it is the GCC default, and the user has to specifically disable them by using the *-fomit-frame-pointer* GCC option. It can be also useful to build the application using debug information (*gcc -g*), to be able to better decode core dumps generated by **GMM**. To show how **GMM** helps in the forensic analysis of a vulnerable software, the above listed sample application is faulted with and without the **GMM** interception, and results coming from a GDB[11] dump are compared. This is the output generated by GDB when **GMM** is not in the loop:

```
[davide@bigblue test]$ gdb -c core.14600 gmm-test
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.
Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
Core was generated by `./gmm-test
uiyueiwyeowyreoywoeywoewoyrowuruowyeouwyeoqwyeowyeoyruowuoewueywewou'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x776f6579 in ?? ()
(gdb) bt
#0  0x776f6579 in ?? ()
Cannot access memory at address 0x77716f65
(gdb)
```

As you can easily guess, the stack has been completely wiped out and both the back trace and the parameters inspection is impossible. This is definitely not good since it does not offer any ground for a sane forensic analysis. The above case is a case where there was not exploit, but instead there was a simple non-malicious buffer overflow. In case of an exploit, things are not much different since the content of the frame is completely over-written by the injected code. Now let's see how the stack frame looks like when GMM is in the loop and is intercepting the required function:

```
[davide@bigblue test]$ gdb -c core.14604 gmm-test
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.
Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
Core was generated by `./gmm-test
uiyueiwyeowyreoywoeywoewoyrowuruowyeouwyeoqwyeowyeoyruowuoewueywewou'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from ../gmm/.libs/libgmm.so...done.
Loaded symbols for ../gmm/.libs/libgmm.so
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/libdl.so.2...done.
Loaded symbols for /lib/libdl.so.2
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0   0x400185d4 in sprintf (str=0xbffff840 "ÀÂ", fmt=0x8048464 "%s") at gmm.c:243
243  WRAP_VFUNCTION(int, sprintf, (char *str, const char *fmt, ...), fmt, (str, fmt, args))
(gdb) bt
#0  0x400185d4 in sprintf (str=0xbffff840 "ÀÂ", fmt=0x8048464 "%s") at gmm.c:243
#1  0x08048376 in do_overflow (str=0xbffffa44
"uiyueiwyeowyreoywoeywoewoyrowuruowyeouwyeoqwyeowyeoyruowuoewueywewouew") at gmm-test.c:33
#2  0x08048396 in run_test (str=0xbffffa44
"uiyueiwyeowyreoywoeywoewoyrowuruowyeouwyeoqwyeowyeoyruowuoewueywewouew") at gmm-test.c:41
#3  0x080483b3 in main (argc=2, argv=0x390d) at gmm-test.c:51
#4  0x40043ab7 in __libc_start_main () from /lib/libc.so.6
(gdb)
```

This looks a lot better, isn't it? The full frame information are available for a full forensic analysis and for a thorough report to be compiled. Not only the back trace, the function parameters and the local variables are available, but also the data used by the attacker is inspect able. Many times indeed, even knowing the exact location of the faulting code, does not allow the developer to understand where the code itself can be broken. And having the data that triggers the fault is of vital importance for debugging the application.

Using And Extending GMM

Like it has been specified in the previous chapter, **GMM** uses the *LD_PRELOAD* capability of the GLIBC loader to insert itself between the application and the GLIBC library. For example, suppose your application named *gmm_target* is an Internet server and it is experiencing weird crashes, or more in general, application or system logs reveal something strange about the application itself. To use **GMM** you first have to build it on your system by downloading the source package from the **GMM** home page. A typical build and install procedure goes like:

```
$ ./configure –prefix=/usr
$ make
$ su
# make install
```

The above sequence of commands will build and install **GMM** libraries and include files. To use GMM you have now to add the path of its shared library inside the LD_PRELOAD environment variable. This can be done either by setting it inside the server start/stop script by doing:

```
export LD_PRELOAD=/usr/lib/libgmm.so:$ LD_PRELOAD
```

Or by specifying it in the same command line used to run the application:

```
$ LD_PRELOAD=/usr/lib/libgmm.so:$ LD_PRELOAD gmm_target ...
```

If the **GMM** interception layer will find something that will overwrite stack frame return addresses, a segmentation fault will be triggered after the saved stack frame is restored. The resulting core file will have all the information needed to successfully debug the target application. Call trace, function parameters and local variables are preserved by the **GMM** interception layer. In case the core file will end up having a corrupted stack frame, it will usually report the name of the function where the fault (not controlled by **GMM**) happened because of a bad return address injected by the attacker. This is likely the name of a function that is not intercepted by **GMM**, and you would need to add such function to the **GMM** monitor. This can be achieved in two different ways depending on the fact that the faulting function is inside GLIBC (or any dynamic library linked by *gmm_target*) or inside the *gmm_target* source code itself. In case the faulting function resides inside GLIBC or any dynamically loaded shared library, you will need to edit the *gmm.c* file of the GMM source distribution to wrap the function with macros defined inside the *gmm.h* include file. In case the function is instead defined inside the *gmm_target* application itself, you need to wrap your function with one of the macros defined inside *gmm.h*. Suppose a function "*int do_this(char *str)*" that needs to be wrapped with **GMM** code. The following changes should be applied to the *gmm_target* source code. First, the existing function should be renamed to "*int do_this_gmm(char *str)*". Then, the **GMM** wrapper macro should be used to define the new instrumented function like:

```
WRAP_USER_FUNCTION(int, do_this, (char *str), (str))
```

The *gmm_target* application should then be rebuilt and it should be run again using the **GMM** monitor. When **GMM** intercept an attempt to over-write the return address that is stored on the stack frame, it will generate e segmentation fault, after having restored the original frame. It is possible to make **GMM** to perform extra actions upon buffer overflow detection. By defining the environment variable *GMM_FAULT_EXEC*, it is possible to ask **GMM** to execute the binary whose path is stored inside such variable before going to generate the segmentation fault. Only one parameter will be passed to the external binary, and this is the process ID (PID) of the faulting task. The GMM fault code will do a *waitpid()* on the child process, so the segmentation fault will not be triggered until the executed process will terminate. The external binary execution feature can be used to perform the more disparate tasks. From sending a notification message to some administrative email address, to running a debugger by attaching the faulting process.

Software

GMM is released under GPL license[12] and the full source is available inside its home page at:

http://www.xmailserver.org/gmm.html

Conclusion

GMM born from a need of the author to inspect a suspect buffer overflow vulnerability inside an existing Open Source application. While existing methods were able to detect and stop the possible malicious code, forensic analysis was impossible due stack frame corruption generated by the buffer overflow. **GMM** helped a lot in finding that, in this particular case, there was no exploit being attempted but it was simply a bug triggered by some special data. All software engineers and system administrators dealing with the debug of faulting applications due possible buffer overflow exploits, will find in **GMM** a precious inspection tool.

References

1. 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 Proceedings of the 7th USENIX Security Conference, 1998.
2. Arash Baratloo, Timothy Tsai, and Navjot Singh. Transparent run-time defense against stack smashing attacks. In Proceedings of the USENIX Annual Technical Conference, June 2000.
3. Exec_shield home page http://people.redhat.com/mingo/exec-shield
4. StackGuard home page http://www.immunix.org/stackguard.html
5. StackShield home page http://www.angelfire.com/sk/stackshield
6. GCC home page http://gcc.gnu.org
7. LibSafe home page http://www.research.avayalabs.com/project/libsafe
8. GLIBC home page http://www.gnu.org/software/libc/libc.html
9. PreLink home page ftp://people.redhat.com/jakub/prelink
10. GMM home page http://www.xmailserver.org/gmm.html
11. GDB home page http://sources.redhat.com/gdb
12. GPL license http://www.gnu.org/copyleft/gpl.html