Buffer overflows: attacks and protections

Cornel Izbaşa*

IeAT, UVT
cornel@ieat.ro

Abstract

This work aims to review the most common vulnerabilities of information systems, explore the attacks based on buffer overflows and some existing protections against this type of attack, and also the implementation of an exploit and a stack-guarding library.

Vulnerabilities of information systems My paper aims to review the possible physical vulnerabilities - attacks on the systems' physical security, cryptographic vulnerabilities - attacks on the encryption algorithms, "channeling" problems - things like format strings vulnerabilities, SQL injection vulnerabilities and buffer overflows, information leaks - social engineering, *man in the middle* etc., *sniffing*, DoS attacks - denial of service attacks, and separately, again buffer overflows - a more in-depth discussion.

Buffer overflows Buffer overflows (see also [1]) are now widely known in the security community. In 2001, 51% of the known-vulnerabilities were based on buffer overflows - especially stack overflows. Overflowing a buffer means writing past the limits of that buffer - overstuffing it with information. The effect is that some data beyond the limits of the buffer gets overwritten as well. If it's simply data, this may or may not be a critical issue, but if it is some kind of metadata - like a return address from the current function or a saved frame pointer - then the attacker might be able to take control of the execution, effectively running arbitrary code with the privileges of the vulnerable program.
Buffer overflows come in two varieties on IA32 - stack-based overflows and heap-based overflows. For the first variety the overflow buffer is in the stack, whereas for the second it is in the heap.
Buffer overflows can be viewed as channeling problems because their exploitation is based on the fact that, on IA32 and similar architectures, the stack is used for passing arguments to functions and for storing local non-static variables but also for storing meta-data, that serve in controlling the execution of the program - and so the stack becomes a channel that is used for passing both data and control codes. In the case of heap overflows this holds as well, since usually they are based on overwriting some pointer to a function which gets called later in the execution.
This is why I think that the best way to get rid of stack-based buffer overflows would be a better architecture for the CPU - namely separate stacks for data and meta-data. Function arguments and non-static local variables are to be held on the data stack, and return addresses and saved frame pointers (which are pointers to the old base of the stack) are to be held on the meta-data stack, that can only be altered by the instructions `call` and `ret` - possibly `leave`. `call` would be pushing the return address and the frame pointer on the meta-data stack, and `leave` and `ret` would recover them.

The exploit I've developed an exploit for a stack-based buffer overflow vulnerability in `xlock`, part of `XFree-4.2.0` that comes with `Redhat 7.2`. It is based on the similar work done for `Slackware 8.1` by **dcrypter && tarranta / oC**.
The overflow buffer consists of a first part filled with NOPs (0x90 on IA32), then the shellcode -

*The author thanks the **IeAT** and Assistant Professor Dr. Marius Minea for supporting this work.

the machine code that spawns a shell on the attacked system if successful and finally a third part consisting of the repeated return address that points into the buffer that we are overflowing. The overflow buffer is copied into the environment variable `XLOCALEDIR` which is handled improperly by `xlock` and some other software part of `XFree-4.2.0`, and then `xlock` is executed to trigger the vulnerability.
The shellcode is almost identical to that used by dcrypter && tarranta, except for a slight optimization in zero-ing the `EAX` register, and a certain small modification that prevents the actual functioning of the exploit due to legal reasons.

Protections A lot of protections have been developed against buffer overflows. Some are based on enforcing non-executable pages or segments (like `PaX`([4]) or `Openwall`), some use a "canary" value that's pushed on the stack after the return address and frame pointer, with the intent to guard these against modification via stack-based overflows (`StackGuard` - see also [5]), others maintain alternative stacks for return addresses and saved frame pointers (`StackShield` and my own `Libprotect`), yet others replace some "dangerous" functions (like `strcpy`, `gets` etc.) with secure versions via a preloading mechanism (`Libsafe`), while `CCured`([3]) classifies the pointers in a C program, inserting bounds-checking code for the "dangerous" ones which guarantees that the resulting program is completely memory-safe.

Libprotect `Libprotect` guards against stack-based overflows and other techniques that attempt to corrupt the return address and saved frame pointer of a function by copying the portion of stack that contains those items into an alternative stack. If there's not more room on the alternative stack (which should never happen for real-world programs) then its "top" pointer is incremented to account for the `push`es but nothing is stored (this was inspired by a related protection - `StackShield`). The advantage of `libprotect` is that it should work on any UNIX on IA32, and possibly even some non-IA32 flavors, since it doesn't use ANY assembly, just plain C. It achieves this by inserting a local variable on the stack of each function right after the saved frame pointer and return address, and uses that as reference for the copying into the alternate stack. Another advantage is that it is programmable - so you can decide which function you wish to protect and how much of the stack you want protected.
In the future, I hope it will use a more advanced memory device for the alternate stack or that the CPU manufacturers will switch to the separate stack architecture described earlier which has also been suggested in [2].

References

[1] Aleph1, *Smashing the stack for fun and profit*, PHRACK, vol. 7, no. 49, 1996.

[2] John P. McGregor and David K. Karig and Zhijie Shi and Ruby B. Lee, *A Processor Architecture Defense against Buffer Overflow Attacks*, 2003.

[3] George Necula and Scott McPeak and Westley Weimer and Matthew Harre n and Jeremy Condit, *CCured documentation*, 2004.

[4] The **PaX** project, `PAGEEXEC.TXT`, 2003.

[5] Gerardo Richarte, *Four different tricks to bypass StackShield and StackGuard protection*, 2002.