

Improving computer security through surreptitious engineering

Mark Feldman
Network Associates Laboratories
Mark_Feldman@nai.com

Engineering is a process that allows us to build things and have confidence in their properties based on the well-understood properties of components and their assembly. An idea becomes a rough sketch becomes a detailed plan. Each step adds specifications and clarifies what came previously. Clarifications and specifications are based on properties of components -- the loading capacity of steel beams, shear-strength of rivets, electrical characteristics of various wire gauges, MTBF. The result is a plan with sufficient detail (architectural drawings, schematics, etc.) to enable those skilled in the field to build the widget or system in question and have confidence in its ability to perform as specified.

Software development often short-circuits the engineering process, resulting in systems that cannot be understood or relied upon. Ideas can be implemented as code without the intervening, rigorous engineering steps. Real-world, physical systems require real-world, physical components. The components have both known properties and associated costs. Assembly is often a multi-step process that requires skilled workers with specialized tools. Software can be written with nothing more than an idea, a text editor, and a compiler. Since the initial ideas have not been clarified, the code is unlikely to be necessary and sufficient to operate properly with correct input and even less so when the input is incorrect or imagined but undocumented constraints are not met. While an idea translated directly to code can operate correctly, the tools at the programmer's hand does not aid in the correct implementation of the idea. The 'C' programming language, still used for many programming projects, provides very little structure to ensure that programs fit their underlying specifications. For example, it does not enforce type-safety, nor does it perform any run-time checking on computed values.

One of the tenets of computer security is that of least privilege. Programs should be vested with the

least amount of privilege necessary to get the job at hand done. That way, should the program be successfully attacked (e.g., replaced through a buffer overflow), the attacker will not be granted excess privilege. On a UNIX-like system, privilege is often all or nothing. In order to bind to a low-numbered port or run as a daemon and then act on individual user's behalf, a program must run as root, the UNIX super-user. Mail, FTP, and other server software often runs as root. These servers are then security-critical software because they run with privileges. Should an attacker compromise them, say through a buffer overflow, the attacker will have root privilege and will be all-powerful. Because of the amount of functionality in modern Mail and FTP servers, they are large, complex programs. In concert with the fact that most are written in 'C,' it is hard to assure that they are free from bugs that could be used by an attacker.

Properly implementing least privilege in a server program is a hard problem, both in terms of designing a solution and properly implementing it. One solution to all-or-nothing privileges is partitioning processes into privileged and unprivileged parts. The unprivileged part contains most program functionality and makes requests for privileged actions to the privileged part, which contains the minimal code required to safely execute the privileged operations. These requests can be audited and permitted by program-specific tests. This limits the program to the privileges actually required. Such process-splitting techniques are employed in an ad hoc manner by some security-critical programs (e.g., OpenSSH). Before implementing privilege partitioning, a far simpler exercise would be to write a program in 'C' that accepts two numbers as input and outputs their sum:

```
#include <stdlib.h>
main () {
    int i, j;
```

```

scanf ("%d %d", &i, &j);
printf ("%d", i+j);
}

```

This program will add 1 and 1 and produce 2 and work for many similar values. Using the GNU C Compiler (GCC) version 2.95.3-2 on the Intel x86 architecture, the above program produces the following results without any errors or warnings:

input (as i)	input (as j)	output
1	1	2
1000000000	1000000000	2000000000
2000000000	2000000000	-294967296
4000000000	4000000000	-2
1	1.1	2
1.1	1	436340273

The program has problems if either of the input numbers or their sum is in excess of the maximum integer the local system can represent. Further, floating-point numbers, which were not part of the initial specification and may just not have been considered, are not handled properly. This is just about as simple a program as can be written and there is the possibility for erroneous output. At the same time, security vulnerabilities related to integer addition and subtraction are becoming popular. They're the latest source of buffer overflows!

As programs increase in complexity, with conditional logic based on input, variable-sized input, and richer data structures, so does the potential for a poorly specified idea being implemented in a way which will permit unexpected input to produce unintended behavior. Buffer overflows are a prime example of unintended behavior resulting from poor specification and implementation. What makes this story worse is that this is how security-critical software is often written. Minimal engineering, unsafe languages, and a lack of well-understood, reusable components. Given this state, what can be done to increase the level of protection associated with security-critical software such as the privilege partitioning scheme?

Improving languages, starting with type safety, would be beneficial, but given the amount of legacy software written in unsafe languages, the languages currently in the comfort zone of

programmers, and the pervasiveness of compilers for unsafe languages, language change is a long-term effort. Further, type safety alone, while enforcing some semantics, does not ensure a strong enough mapping between idea and implementation to prevent gaps. We must provide tools that both improve security and make the programmer's life easier. A partial solution is to engineer components – libraries – that programmers want to use. Libraries that provide valuable security services using best practices, are understandable, and lead to increased confidence. This is a path we are following. Our first library deals with the pervasive problem of excess privilege in programs that run as servers.

The Privman* library simplifies the partitioning of processes into privileged and non-privileged parts. The primary benefit of the Privman library is that it is a systematic, reusable framework and library for developing partitioned programs. We have demonstrated the feasibility of the approach in two example systems: tthttpd and WU-FTPD. WU-FTPD, a feature-rich FTP server, has been the target of successful attacks. While it is possible that there are additional latent flaws in WU-FTPD that might become avenues for attack, versions of WU-FTPD using Privman will not give a successful attacker root privilege. Privman makes least privilege palatable to the programmer. Implementing least privilege with Privman is far easier than doing so from scratch, and it is designed to be secure, easily reviewed, and easily understood.

We are looking into similar, library-based components that have understandable behavior themselves and therefore increase the understandability and confidence in the security aspects of the programs using them. As the quantity of such components and their use increases, security-critical aspects will continue to migrate from large, feature-rich programs that cannot be assured to work correctly into components that can be assured. Just don't tell the computer scientists that it's engineering.

* Privman is available at <http://opensource.nailabs.com/privman/>