September 1995

Classic of the month

Reflections on Trusting Trust

Ken Thompson

Introduction

I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX swept into popularity with an industry-wide change from entral main frames to autonomous minis. I suspect that Daniel Bobrow (1) would be here instead of me if he could not afford a PDP-10 and ad had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of miscoordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed.

I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

Stage I

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to demonstrate skill and determine a winner.

Figure I shows a self-reproducing program in the C programming language. (The purist will note that the program is not precisely a self-reproducing program, but will produce a self-reproducing program.) This entry is much too large to win a prize, but it demonstrates the technique and has two important properties that I need to complete my story: (I) This program can be easily written by another program. (2) This pro- gram can contain an arbitrary amount of excess baggage that will be reproduced along with the main algorithm. In the example, even the comment is reproduced.

Stage II

The C compiler is written in C. What I am about to describe is one of many "chicken and egg" problems that arise when compilers are written in their own language. In this ease, I will use a specific example from the C compiler.

C allows a string construct to specify an initialized character array. The individual characters in the string ean be escaped to represent unprintable characters. For example,

    "Hello world\n"

represents a string with the eharaeter "\n," representing the new line character.

Figure 2 is an idealization of the code in the C compiler that interprets the character escape sequence. This is an amazing piece of code. It "knows" in a completely portable way what character code is compiled for a new line in any character set. The act of knowing then allows it to recompile itself, thus perpetuating the knowledge.

Suppose we wish to alter the C compiler to include the sequence "\v" to represent the vertical tab character. The extension to Figure 2 is obvious and is presented in Figure 3. We then recompile the C compiler, but we get a diagnostic. Obviously, since the binary version of the compiler does not know about "\v," the source is not legal C. We must "train" the compiler. After it "knows" what "\v" means, then our new change will become legal C. We look up on an ASCII chart that a vertical tab is decimal 11. We alter our souree to look like Figure 4. Now the old compiler accepts the new source. We install the resulting binary as the new official C compiler and now we can write the portable version the way we had it in Figure 3.

This is a deep concept. It is as close to a "learning" program as I have seen. You simply tell it once, then you can use this self-referencing definition.

Stage III

Again, in the C compiler, Figure 5 represents the high-level control of the C compiler where the routine "compile" is called to compile the next line of source. Figure 6 shows a simple modification to the compiler that will deliberately miscompile source whenever a particular pattern is matched. If this were not deliberate, it would be called a compiler "bug." Since it is deliberate, it should be called a "Trojan horse."

The actual bug I planted in the compiler would match code in the UNIX "login" command. The replacement code would miscompile the login command so that it would accept either the intended encrypted password or a particular known password. Thus if this code were installed in binary and the binary were used to compile the login command, I could log into that system as any user.

Such blatant code would not go undetected for long. Even the most casual perusal of the source of the C compiler would raise suspicions.

The final step is represented in Figure 7. This simply adds a second Trojan horse to the one that already exists. The second pattern is aimed at the C compiler. The replacement code is a Stage I self-reproducing program that inserts both Trojan horses into the compiler. This requires a learning phase as in the Stage II example. First we compile the modified source with the normal C compiler to produce a bugged binary. We install this binary as the official C. We can now remove the bugs from the source of the compiler and the new binary will reinsert the bugs whenever it is compiled. Of course, the login command will remain bugged with no trace in source anywhere.

Moral

The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code. In demonstrating the possibility of this kind of attack, I picked on the C compiler. I could have picked on any program-handling program such as an assembler, a loader, or even hardware microcode. As the level of program gets lower, these bugs will be harder and harder to detect. A well installed microcode bug will be almost impossible to detect.

After trying to convince you that I cannot be trusted, I wish to moralize. I would like to criticize the press in its handling of the "hackers," the 414 gang, the Dalton gang, etc. The acts performed by these kids are vandalism at best and probably trespass and theft at worst. It is only the inadequacy of the criminal code that saves the hackers from very serious prosecution. The companies that are vulnerable to this activity (and most large companies are very vulnerable) are pressing hard to update the criminal code. Unauthorized access to computer systems is already a serious crime in a few states and is currently being addressed in many more state legislatures as well as Congress.

There is an explosive situation brewing. On the one hand, the press, television, and movies make heros of vandals by calling them whiz kids. On the other hand, the acts performed by these kids will soon be punish- able by years in prison.

I have watched kids testifying before Congress. It is clear that they are completely unaware of the seriousness of their acts. There is obviously a cultural gap. The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor's house. It should not matter that the neighbor's door is unlocked. The press must learn that misguided use of a computer is no more amazing than drunk driving of an automobile.

Acknowledgment

I first read of the possibility of such a Trojan horse in an Air Force critique (4) of the security of an early implementation of Multics. I can- not find a more specific reference to this document. I would appreciate it if anyone who can supply this reference would let me know.

References

1. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L., and Tomlinson, R.S. TENEX, a paged time-sharing system for the PDP-IO. Commun. ACM 15, 3(Mar. 1972), 135-143.

2. Kernighan, B.W., and Ritchie, D.M. The C Programming Language. Prentice-Hall, Englewood Cliffs, N.J., 1978.

3. Ritchie, D.M., and Thompson, K. The UNIX time-sharing system. Commun. ACM 17, 7(July 1974), 365-375.

4. Unknown Air Force Document.