Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

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[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had 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.

[1] UNIX is a trademark of AT&T Bell Laboratories.

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 1 shows a self-reproducing program in the C[3] 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: 1) This program can be easily written by another program. 2) This program 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.

```
char s[ ] = {
        '\t',
        '0',
        '\n',
        '}',
        ';',
        '\n',
        '\n',
        '/',
        '*',
        '\n',
        (213 lines deleted)
        0
};

/*
 * The string s is a
 * representation of the body
 * of this program from '0'
 * to the end.
 */

main( )
{
        int i;

        printf("char\ts[ ] = {\n");
        for(i=0; s[i]; i++)
                printf("\t%d, \n", s[i]);
        printf("%s", s);
}
```

Here are some simple transliterations to allow a non-C programmer to read this code.

=	assignment
==	equal to .EQ.
!=	not equal to .NE.
++	increment
'x'	single character constant
"xxx"	multiple character string
%d	format to convert to decimal
%s	format to convert to string
\t	tab character
\n	newline character

FIGURE 1.

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 case, 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 can be escaped to represent unprintable characters. For example,

"Hello world\n"

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

Figure 2.1 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 2.1 is obvious and is presented in Figure 2.2. 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 source to look like Figure 2.3. 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 2.2.

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 3.1 represents the high level control of the C compiler where the routine "com-

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
...
```

FIGURE 2.2.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
if(c == 'v')
        return('\v');
...
```

FIGURE 2.1.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\ n');
if(c == 'v')
        return(11);
...
```

FIGURE 2.3.

pile" is called to compile the next line of source. Figure 3.2 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 3.3. 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.

```
compile(s)
char *s;
{
        ...
}
```

FIGURE 3.1.

```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

FIGURE 3.2.

```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        ...
}
```

FIGURE 3.3.

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 punishable 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 cannot 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-10. *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*, (July 1974), 365–375.
4. Unknown Air Force Document.

Author's Present Address: Ken Thompson, AT&T Bell Laboratories, Room 2C-519, 600 Mountain Ave., Murray Hill, NJ 07974.

Thirty Years Later: Lessons from the Multics Security Evaluation

Paul A. Karger
IBM Corp., T. J. Watson Research Center
Yorktown Heights, NY 10598
karger@watson.ibm.com

Roger R. Schell
Aesec Corporation
Pacific Grove, CA 93950
schellr@ieee.org

ABSTRACT

Almost thirty years ago a vulnerability assessment of Multics identified significant vulnerabilities, despite the fact that Multics was more secure than other contemporary (and current) computer systems. Considerably more important than any of the individual design and implementation flaws was the demonstration of subversion of the protection mechanism using malicious software (e.g., trap doors and Trojan horses). A series of enhancements were suggested that enabled Multics to serve in a relatively benign environment. These included addition of "Mandatory Access Controls" and these enhancements were greatly enabled by the fact the Multics was designed from the start for security. However, the bottom-line conclusion was that "restructuring is essential" around a verifiable "security kernel" before using Multics (or any other system) in an open environment (as in today's Internet) with the existence of well-motivated professional attackers employing subversion. The lessons learned from the vulnerability assessment are highly applicable today as governments and industry strive (unsuccessfully) to "secure" today's weaker operating systems through add-ons, "hardening", and intrusion detection schemes.

1 INTRODUCTION

In 1974, the US Air Force published [24] the results of a vulnerability analysis of what was then the most secure operating system available in the industry. That analysis of Multics examined not only the nature and significance of the vulnerabilities identified but also the technology and implementation prospects for effective solutions. The results are still interesting today, because many of the concepts of UNIX and other modern operating systems came directly from Multics.

This paper revisits those results and relates them to the widespread security problems that the computer industry is suffering today. The unpleasant conclusion is that although few, if any, fundamentally new vulnerabilities are evident today, today's products generally do not even include many of the Multics security techniques, let alone the enhancement identified as "essential."

2 Multics Security Compared to Now

Multics offered considerably stronger security than most systems commercially available today. What factors contributed to this?

2.1 Security as a Primary Original Goal

Multics had a primary goal of security from the very beginning of its design [16, 18]. Multics was originally conceived in 1965 as a computer utility – a large server system on which many different users might process data. This requirement was based on experience from developing and running the CTSS time sharing system at MIT. The users might well have conflicting interests and therefore a need to be protected from each other – a need quite like that faced today by a typical Application Service Provider (ASP) or Storage Service Provider (SSP). With the growth of individual workstations and personal computers, developers concluded (incorrectly) that security was not as important on minicomputers or single-user computers. As the Internet grew, it became clear that even single-user computers needed security, but those intervening ill-founded decisions continue to haunt us today. Today, the computer utility concept has returned [13], but today's operating systems are not even up to the level of security that Multics offered in the early 1970s, let alone the level needed for a modern computer utility. There has been work on security for Computational Grids [12, 21], but this work has focused almost entirely on authentication and secure communication and not on the security of the host systems that provide the utility.

2.2 Security as Standard Product Feature

One of the results of the US Air Force's work on Multics security was a set of security enhancements [44] that ultimately became the primary worked example when defining the standard for Class B2 security in the Trusted Computer System Evaluation Criteria (TCSEC) [2], better known as the Orange Book. In the years that followed, other similar enhancements [9, 11] were done for other operating systems, generally targeted for the much weaker Class B1 level, but none of these security enhanced operating systems integrated as well with applications as the Multics enhancements had. This was because unlike ALL of the other security enhanced systems, the Multics

security enhancements were made part of the standard product, shipped to ALL users, rather than only to those users who specifically requested them. A significant reason why security enhancements to other operating systems were not integrated into the mainstream products was that, unlike Multics, neither the operating systems nor the applications generally were well structured for security, making security additions awkward.

While this difference might not seem significant initially, and indeed might even seem detrimental to the manufacturer's revenue stream (since security enhancements could command high prices), the reality was very different. Because the Multics security enhancements, including mandatory access controls, were shipped to ALL customers, this meant that the designers of applications had to make sure that their applications worked properly with those controls. By contrast, many application developers for other systems with optional security enhancements don't even know that the security enhancement options exist, let alone develop applications that work with them.

2.3 No Buffer Overflows

One of the most common types of security penetrations today is the buffer overflow [6]. However, when you look at the published history of Multics security problems [20, 28-30], you find essentially no buffer overflows. Multics generally did not suffer from buffer overflows, both because of the choice of implementation language and because of the use of several hardware features. These hardware and software features did not make buffer overflows impossible, but they did make such errors much less likely.

2.3.1 Programming in PL/I for Better Security

Multics was one of the first operating systems to be implemented in a higher level language.[1] While the Multics developers considered the use of several languages, including BCPL (an ancestor of C) and AED (Algol Extended for Design), they ultimately settled on PL/I [15].

Although PL/I had some influence on the development of C, the differences in the handling of varying length data structures between the two languages can be seen as a major cause of buffer overflows. In C, the length of all character strings is varying and can only be determined by searching for a null byte. By contrast, PL/I character strings may be either fixed length or varying length, but a maximum length must always be specified, either at compile time or in an argument descriptor or in another variable using the REFER option. When PL/I strings are used or copied, the maximum length specifications are honored by the compiled code, resulting in automatic string truncation or padding, even when full string length checking is not enabled. The net result is that a PL/I programmer would have to work very hard to program a buffer overflow error, while a C programmer has to work very hard to avoid programming a buffer overflow error.

Multics added one additional feature in its runtime support that could detect mismatches between calling and called argument descriptors of separately compiled programs and raise an error.

PL/I also provides richer features for arrays and structures. While these differences are not as immediately visible as the character string differences, an algorithm coded in PL/I will have less need for pointers and pointer arithmetic than the same algorithm coded in C. Again, the compiler will do automatic truncation or padding, even when full array bounds checking is not enabled.

While neither PL/I nor C are strongly typed languages and security errors are possible in both languages, PL/I programs tend to suffer significantly fewer security problems than the corresponding C programs.

2.3.2 Hardware Features for Better Security

Multics also avoided many of the current buffer overflow problems through the use of three hardware features. First and most important, Multics used the hardware execute permission bits to ensure that data could not be directly executed. Since most buffer overflow attacks involve branching to data, denying execute permission to data is a very effective countermeasure. Unfortunately, many contemporary operating systems have not used the execute permission features of the x86 segmentation architecture until quite recently [17].

Second, Multics virtual addresses are segmented, and the layout of the ITS pointers is such that an overflow off the end of a segment does not carry into the segment number portion. The net result is that addressing off the end of a segment will always result in a fault, rather than referencing into some other segment. By contrast, other machines that have only paging (such as the VAX, SPARC, or MIPS processors) or that allow carries from the page offset into the segment number (such as the IBM System/370) do not protect as well against overflows. With only paging, a system has to use no-access guard pages that can catch some, but not all references off the end of the data structure. In the case of the x86 processors, although the necessary segmentation features are present, they are almost never used, except for operating systems specifically designed for security, such as GEMSOS [32].

Third, stacks on the Multics processors grew in the positive direction, rather than the negative direction. This meant that if you actually accomplished a buffer overflow, you would be overwriting unused stack frames, rather than your own return pointer, making exploitation much more difficult.

2.4 Minimizing Complexity

Multics achieved much of its security by structuring of the system and by minimizing complexity. It is interest-

1 Burroughs' use of ALGOL for the B5000 operating system was well known to the original Multics designers.

ing to compare the size and complexity of Multics of that time with current systems, such as the NSA's Security Enhanced Linux (SELinux). As documented in [41], the ring 0 supervisor of Multics of 1973 occupied about 628K bytes of executable code and read only data. This was considered to be a very large system. By comparison, the size of the SELinux module with the example policy [37] code and read only data has been estimated [22] to be 1767K bytes. This means that just the example security policy of SELinux is more than 2.5 times bigger than the entire 1973 Multics kernel and that doesn't count the size of the Linux kernel itself. These numbers are quite inexact, but they raise a warning. Given that complexity is the biggest single enemy of security, it is important that the SELinux designers examine whether or not there is a complexity problem to be addressed.

3 Malicious Software

One of the major themes of the Multics Security Evaluation was to demonstrate the feasibility of malicious software attacks. Sadly, we were all too successful, and the predictions of the report have been very prophetic.

At that time, we hypothesized that professional penetrators would find that distribution of malicious software would prove to be the attack of choice. In the 1970s with the Cold War raging, our assumption was that the most immediate professional penetrators would be foreign espionage agents and that the defense establishment would be the target of choice, but we expected commercial penetrators to follow. Of course, malicious software (viruses, worms, Trojan horses, etc.) is very common on the Internet today, targeting commercial and university interests as well as the defense establishment. For example, **The New York Times** has reported [36] on surreptitious add-ons to certain peer-to-peer file sharing utilities that can also divert the revenue stream of commissions paid to affiliates of electronic commerce web sites. While the legal status of such add-ons is unclear, the reaction of affiliates who have had their commissions diverted has been very negative. Anderson [8] shows some additional modern examples.

3.1 Installing Trap Doors

To demonstrate our hypothesis, we not only looked for exploitable flaws in Multics security, but we also attempted to plant trap doors into Multics to see if they could be found during quality assurance and whether they would be distributed to customer sites. Section 3.4.5.2 of the report described a trap door that was installed in the 645 processor, but was not distributed, as the 645 was being phased out. This trap door did demonstrate a property expected of real-world trap doors – its activation was triggered by a password or key that insured the vulnerability was not invoked by accident or discovered by quality assurance or other testing.

Section 3.4.6 gave a brief hint of a trap door that was installed into the 6180 development system at MIT in the routine hcs_$set_ring_brackets_. We verified that the trap door was distributed by Honeywell to the 6180 processor that was installed at the Air Force Data Services Center (AFDSC) in the basement of the Pentagon in an area that is now part of the Metro station. Despite the hint in section 3.4.6, the trap door was not discovered until roughly a year after the report was published. There was an intensive security audit going on at the General Motors Multics site, and the trap door, which was neutered to actually be benign, (it needed a one instruction change to be actively malicious) gave an anomalous error return. The finding of the trap door is described on the Multics web site [45], although the installation of the trap door is incorrectly attributed to The MITRE Corporation.

3.2 Malicious Software Predictions

In addition to demonstrating the feasibility of installing a trap door into commercial software and having the manufacturer then distribute it to every customer in the world, the report also hypothesized a variety of other malicious software attacks that have unfortunately all come true. Section 3.4.5.1 proposed a variety of possible attack points.

3.2.1 Malicious Developers

We suggested that malicious developers might create malicious software. Since that time, we have seen many products with either surreptitious backdoors that allow the developer to gain access to customer machines or with so-called Easter eggs that are simply concealed pieces of code that do humorous things when activated. As expected of malicious trapdoors, activation of most Easter eggs is triggered by a unique key or password that is not likely to be encountered in even extensive quality assurance testing. In most cases, the Easter eggs have NOT been authorized by the development managers, and are good examples of how developers can insert unauthorized code. The primary difference between an Easter egg and a piece of malicious software is the developer's intent. Fortunately for most instances, the developers are not malicious.

3.2.2 Trap Doors During Distribution

The report predicted trap doors inserted into the distribution chain for software. Today, we frequently see bogus e-mail or downloads claiming to contain important updates to widely used software that in fact contain Trojan horses. One very recent incident of this kind is reported in [19].

3.2.3 Boot-Sector Viruses

The report predicted trap doors during installation and booting. Today, we have boot sector viruses that are quite common in the wild.

3.2.4 Compiler Trap Doors

The trap doors described in the report were patches to the binary object files of the system. The report suggested a countermeasure to such object code trap doors by having customers recompile the system from source, although the report also notes that this could play directly into the hands of the penetrator who has made changes in the source code. In fact, the AFDSC Multics contract specifically required that Honeywell deliver source code to the Pentagon to permit such recompilations. However, the report pointed out the possibility that a trap door in the PL/I compiler could install trap doors into the Multics operating system when modules were compiled and could maintain its own existence by recognizing when the PL/I compiler was compiling itself. This recognition was the basis several years later for the TCSEC Class A1 requirement for generation of new versions from source using a compiler maintained under strict configuration control. A recent story on CNET news [14] reports that the Chinese government has similar concerns about planted trap doors.

This suggestion proved an inspiration to Ken Thompson who actually implemented the self-inserting compiler trap door into an early version of UNIX. Thompson described his trap door in his 1984 Turing Award paper [40], and attributed the idea to an "unknown Air Force document," and he asked for a better citation. The document in question is in fact the Multics Security Evaluation report, and we gave a new copy of the report to Thompson after his paper was published. Thompson has corrected his citation in a reprint of the paper [39].

3.3 Auditing and Intrusion Detection

Multics had some limited auditing capabilities (basically recording date time modified (DTM) on files) at the time of the original vulnerability analysis, although a great deal more auditing was added as part of the Multics Security Enhancements project [44]. Section 3.4.4 showed how a professional attacker could bypass the auditing capabilities while installing malicious software.

This has major implications for today, because intrusion detection has become a major area of security product development. Unfortunately, many of the intrusion detection systems (IDSs) that have been developed do not deal well with the possibility of a professional attacker going in and erasing the evidence before it can be reported to an administrator to take action. Most IDSs rely on pattern recognition techniques to identify behaviors that are indicative of an attack. However, a professional penetrator would do as we did – debug the penetration programs on a duplicate machine, the Rome Air Defense Center Multics system in our case, before attacking the desired target, the MIT site in our case. This would give the IDS only a single isolated anomaly to detect, and the IDS would have to respond before the professional attacker could erase the evidence as we did with the Multics DTM data.

Most evaluations of IDS systems have focused on how well the IDS responds to a series of pre-recorded traces. A more relevant evaluation of an IDS system would be to see how it responds to a well-prepared and well-funded professional (but ethical) attacker who is prepared to erase the auditing data and to install trap doors immediately for later exploitation, or subvert the operating system on which the IDS is running to selectively bypass the IDS all together. This type of selectively triggered attack of a limited subsystem (i.e., the IDS) through a subverted host operating system is illustrated by the "key string trigger trap door" in section 3.4.5.1 of the report.

4 What Happened Next?

This section discusses the immediate outcomes from the Vulnerability Analysis.

4.1 Multics Security Enhancements

The primary outcome of the vulnerability analysis, and indeed the reason it was done at all was the Multics security enhancements effort [44]. The Air Force had initially approached Honeywell with a proposal for an R&D effort to improve Multics security for the Air Force Data Services Center, but Honeywell initially declined on the basis that Multics was already sufficiently secure. The vulnerability analysis was conducted to provide evidence of the need for significant enhancements. This security enhanced version of Multics ultimately achieved a Class B2 evaluation from the National Computer Security Center in 1985 [3].

4.2 Publicity

The results of the vulnerability received a certain level of press publicity. **Fortune** magazine published [7] an interview with Steve Lipner of the MITRE Corporation that was a bit sensationalized. It focused on the illegal operator's console message described in section 3.3.4.3. That message quoted from a Honeywell press release that appeared in an internal Honeywell newsletter and a Phoenix, AZ newspaper article[2] that incorrectly stated that the Air Force had found some security problems in Multics, but they were all fixed and that the Air Force now "certified" Multics security. Not only was the Honeywell press release inaccurate, but it had not been cleared for public release, as required in the Multics security enhancements contract. While the **Fortune** article was a bit sensational, the story was then retold in a text book [26] in a version that was barely recognizable to us.

A much better description of the work was done by Whiteside[43], based on his earlier articles for **The New Yorker**. Whiteside interviewed both Schell and Lipner and did a much more accurate account of the effort.

2 The authors have been unable to find copies of these articles and would appreciate any information on how to locate them.

4.3 Multics Kernel Design Project

As the report suggested in section 4.2.2, a project was started to build a security kernel version of Multics, one intended to ultimately meet what was later defined as the requirements of a Class A1 evaluation of the Orange Book [2]. This effort was conducted jointly by Honeywell, MIT, the MITRE Corporation, and the Air Force with results published in: [10, 33-35, 38].

4.4 Direction to Stop Work

In August 1976, Air Force Systems Command directed termination of the Air Force's computer security work, including the Multics Kernel Design Project, despite the fact that the work was quite successful up to that point, had widespread support from user agencies in the Army, the Navy, and the Air Force, and had been by DoD standards, quite inexpensive. The adverse impact of that termination on the DoD is described in [31]. Another view of those events can be seen in [23, pp. II-74 – II-75] and in the GAO report on the termination [4].

5 Dispelling Mythology

The Multics vulnerability analysis got enough publicity that a certain amount of mythology developed about what happened. We hope this section can dispel at least some of that mythology.

One story that circulated was that the penetrations were only possible, because the authors had special privileged access to Multics. While both authors had been members of the MIT Multics development team (Roger Schell as a Ph.D. student and Paul Karger as an undergraduate), both had left the Multics team months before the vulnerability analysis was even considered. No privileged accounts on either of those systems were used in the analysis. (Privileged accounts were used on RADC-Multics and at the Air Force Data Services Center, but those were only to show that a penetrator could take advantage of a duplicate system to debug penetrations ahead of time and to verify the distribution of the trap door.) The Air Force purchased computer time on the MIT-Multics system and on Honeywell's Multics system in Phoenix, AZ, but both systems routinely sold computer time. Multics source code was readable by any user on the MIT site, much as source code for open source systems is generally available today.

The entire vulnerability analysis was carried out with a ground rule of not causing any real harm to any of the systems under attack. All the documentation and listings of exploit programs was kept in a safe approved for storing TOP SECRET information (although that particular safe actually had no classified material in it). Decrypted password files were also stored in the safe, to ensure that no user accounts would be compromised. Exploit programs were protected with very restrictive access control lists and kept encrypted at all times, except when actually in use. The applicable contracts specified that the purchased computer time at MIT and Phoenix would be used for security testing and evaluation purposes, so there was no question of the legality of the analysis. Finally, publication of [24] was delayed until after all of the penetrations described were repaired.

For several years after the completion of the vulnerability analysis, the authors were periodically contacted when unexplained system crashes occurred on the MIT site, just in case they had been caused by a remaining hidden trap door. However, as part of the basic policy of the analysis to cause no real harm, no trap doors were left behind after the conclusion of the vulnerability analysis.

6 Security has gotten worse, not better

Today, government and commercial interest in achieving processing and connectivity encompassing disparate security interests is driving efforts to provide security enhancements to contemporary operating systems [27]. These efforts include the addition of mandatory access controls and "hardening" (i.e., the removal of unnecessary services.) Such enhancements result in systems less secure than Multics, and Multics, with its security enhancements, was only deemed suitable for processing in a relatively benign "closed" environment [5]. It was concluded that operation of Multics in an "open" environment would require restructuring around a verifiable security kernel to address the threat of professional attacks using malicious software (e.g., trap doors).

6.1 Weak Solutions in Open Environments

Given the understanding of system vulnerabilities that existed nearly thirty years ago, today's "security enhanced" or "trusted" systems would not be considered suitable for processing even in the benign closed environment. Also, considering the extent of network interconnectivity and the amount of commercial off-the-shelf (COTS) and other software without pedigree (e.g., libraries mined from the Web), today's military and commercial environments would be considered very much "open". To make matters worse, everyone is now expected to be a system administrator of their own desktop system, but without any training!

Furthermore, quality assurance testing and subjecting systems to authorized (ethical) hackers are among the most common approaches for concluding that the security of such systems is adequate. There have even been apparently serious proposals [1] for something as important as voting to use student hackers "to test computerized voting systems". Yet it is evident on the face of it that such approaches are utterly useless for detecting the vulnerability most attractive to a professional – malicious software triggered by a unique key or password unknown to those making this assessment.[3]

3 Note that this does not mean that penetration testing or ethical hacking has no use. One of the authors currently works in IBM's Global Security Analysis Laboratory, an ethical hacking organization. Penetration

Thus, systems that are weaker than Multics are considered for use in environments in excess of what even Multics could deliver without restructuring around a security kernel. There really seem to be only four possible conclusions from this: either (1) today's systems are really much more secure than we claim; (2) today's potential attackers are much less capable or motivated; (3) the information being processed is much less valuable; or (4) people are unwilling or unable to recognize the compelling need to employ much better technical solutions.

6.2 Technical Solutions

Though Multics was never restructured around a security kernel as recommended, the technical ability to structure a full featured commercial operating system around a kernel designed to meet Class A1 while maintaining good operating system and application compatibility was demonstrated for the VAX [25]. Successful fielding on the open Internet of a highly secure system using the popular x86 processor with the commercial NSA-certified Class A1 GEMSOS [32] security kernel for VPN security management has also been demonstrated [42].

In the nearly thirty years since the report, it has been demonstrated that the technology direction that was speculative at the time can actually be implemented and provides an effective solution to the problem of malicious software employed by well-motivated professionals. Unfortunately, the mainstream products of major vendors largely ignore these demonstrated technologies. In their defense most of the vendors would claim that the marketplace is not prepared to pay for a high assurance of security. And customers have said they have never been offered mainstream commercial products that give them such a choice, so they are left with several ineffective solutions collected under marketing titles like "defense in depth".

In our opinion this is an unstable state of affairs. It is unthinkable that another thirty years will go by without one of two occurrences: either there will be horrific cyber disasters that will deprive society of much of the value computers can provide, or the available technology will be delivered, and hopefully enhanced, in products that provide effective security. We hope it will be the latter.

7 Acknowledgments

We must thank Iris Partlow of the IBM Business Support Center for retyping the 1974 report. Very helpful comments came from Mike Thompson of Aesec Corporation, Jeremy Epstein of webMethods, Wietse Venema and David Safford of IBM Research, Steve Sutton of PatientKeeper, and Tom Van Vleck of NAI Labs. Tom also runs the Multics web site (http://www.multicians.org), where we found some of the information needed for this commentary. Tom was the system administrator and a designer of Multics, so we also thank him (nearly 30 years later) for understanding that our work was to make Multics more secure, even if its presentation was considered overly dramatic at times.

testing can identify some vulnerabilities for repair. However, the failure of any particular penetration team to find vulnerabilities does not imply their absence. It only indicates that the team failed to find any.

REFERENCES

1. *Broward Officials Want Students to Try Hacking Mock Election*, in *Assoc. Press* 16 August 2001: Ft. Lauderdale, FL. URL: http://www.wtsp.com/news/2001_08/16_mock_election.htm

2. *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD, December 1985: Washington, DC. URL: http://csrc.nist.gov/publications/history/dod85.pdf

3. *Final Evaluation Report of Multics, MR11.0*, CSC-EPL-85/003 (releasable only to US Government and their contractors), 1 September 1985, National Computer Security Center: Ft. George G. Meade, MD. URL: http://www.radium.ncsc.mil/tpep/library/fers/tcsec_fers.html

4. *Multilevel Computer Security Requirements of the World Wide Military Command and Control System (WWMCCS)*, LCD-78-106, 5 April 1978, General Accounting Office: Washington, DC. URL: http://www.gao.gov/docdblite/summary.php?&rptno=LCD-78-106

5. *Technical Rationale Behind CSC-STD-003-85: Computer Security Requirements -- Guidance for Applying the Department of Defense Trusted Computer System Evaluation Criteria in Specific Environments*, CSC-STD-004-85, 25 June 1985, DoD Computer Security Center: Ft. George G. Meade, MD.

6. Aleph One, *Smashing the Stack for Fun and Profit.* **Phrack**, 8 November 1996. **7**(49). URL: http://www.phrack.org/show.php?p=49&a=14

7. Alexander, T., *Waiting for the Great Computer Rip-off.* **Fortune**, 1974. **XC**(1): p. 142-150.

8. Anderson, E.A., III, *A Demonstration of the Subversion Threat: Facing a Critical Responsibility in the Defense of Cyberspace*, M.S. in *Computer Science* 2002, Naval Postgraduate School: Monterrey, CA. URL: http://theses.nps.navy.mil/Thesis_02Mar_Anderson_Emory.pdf

9. Berger, J.L., J. Picciotto, *et al.*, *Compartmented Mode Workstation: Prototype Highlights.* **IEEE Transactions on Software Engineering**, June 1990. **16**(6): p. 608-618.

10. Biba, K.J., S.R. Ames, *et al.*, *A Preliminary Specification of a Multics Security Kernel*, WP-20119, April 1975, The MITRE Corporation: Bedford, MA.

11. Blotcky, S., K. Lynch, *et al. SE/VMS: Implementing Mandatory Security in VAX/VMS.* in **Proceedings of the 9th National Computer Security Conference**. 15-18 September 1986, Gaithersburg, MD National Bureau of Standards. p. 47-54.

12. Butler, R., V. Welch, *et al.*, *A National-Scale Authentication Infrastructure.* **Computer**, December 2000. **33**(12): p. 60-66.

13. Carr, D.F., *The Hot Hand in the ASP Game.* **Internet World**, 15 January 2000. **6**(2): p. 53. URL: http://www.internetworld.com/magazine.php?inc=011500/1.15cover_story.html

14. Cooper, C., *Who says paranoia doesn't pay off?* **CNET News.com**, 20 September 2002. URL: http://news.com.com/2010-1071-958721.html

15. Corbató, F.J., *PL/I As a Tool for System Programming.* **Datamation**, May 1969. **15**(5): p. 68-76. URL: http://home.nycap.rr.com/pflass/plisprg.htm

16. Corbató, F.J. and V.A. Vyssotsky. *Introduction and Overview of the Multics System.* in **Fall Joint Computer Conference**. 1965, Washington, DC Vol. AFIPS Conference Proceedings Vol. 27. Spartan Books. p. 185-196. URL: http://www.multicians.org/fjcc1.html

17. Cowan, C., P. Wagle, *et al. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade.* in **DARPA Information Survivability Conference and Expo (DISCEX)**. 25-27 January 2000, Hilton Head Island, SC Vol. 2. p. 119-129. URL: http://www.immunix.org/StackGuard/discex00.pdf

18. David, E.E., Jr. and R.M. Fano. *Some Thoughts About the Social Implications of Accessible Computing.* in **Fall Joint Computer Conference**. 1965, Washington, DC Vol. AFIPS Conference Proceedings Vol. 27. Spartan Books. p. 243-247. URL: http://www.multicians.org/fjcc6.html

19. Dougherty, C., *Trojan Horse OpenSSH Distribution*, CERT® Advisory CA-2002-24, 2 August 2002, CERT Coordination Center, Carnegie Mellon University: Pittsburgh, PA. URL: http://www.cert.org/advisories/CA-2002-24.html

20. Forsdick, H.C. and D.P. Reed, *Patterns of Security Violations: Multiple References to Arguments*, in *Ancillary Reports: Kernel Design Project*, D.D. Clark, Editor, June 1977, MIT/LCS/TM-87, Laboratory for Computer Science, Massachusetts Institute of Technology: Cambridge, MA. p. 34-49.

21. Foster, I., C. Kesselman, *et al. A Security Architecture for Computational Grids.* in **Proceedings of the 5th ACM Conference on Computer and Communications Security**. 2-5 November 1998, San Francisco, CA p. 83-92.

22. Jaeger, T., *Personal Communication*, 30 July 2002, IBM Corp., T. J. Watson Research Center: Hawthorne, NY.

23. Jelen, G.F., *Information Security: An Elusive Goal*, P-85-8, June 1985, Program on Information Resources Policy, Harvard University: Cambridge, MA. URL: http://www.pirp.harvard.edu/publications/pdf-blurb.asp?id=238

24. Karger, P.A. and R.R. Schell, *Multics Security Evaluation: Vulnerability Analysis*, ESD-TR-74-193, Vol. II, June 1974, HQ Electronic Systems Division: Hanscom AFB, MA. URL: http://csrc.nist.gov/publications/history/karg74.pdf

25. Karger, P.A., M.E. Zurko, *et al.*, *A Retrospective on the VAX VMM Security Kernel.* **IEEE Transactions on Software Engineering**, November 1991. **17**(11): p. 1147-1165.

26. Logsdon, T., *Computers & Social Controversy*, 1980, Computer Science Press: Potomoc, MD. p. 170-175, 185.

27. Loscocco, P. and S. Smalley. *Integrating Flexible Support for Security Policies into the Linux Operating System.* in **Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)**. 2001, Boston, MA. URL: http://www.nsa.gov/selinux/doc/freenix01.pdf

28. Saltzer, J.H., *Repaired Security Bugs in Multics*, in *Ancillary Reports: Kernel Design Project*, D.D. Clark, Editor, June 1977, MIT/LCS/TM-87, Laboratory for Computer Science, Massachusetts Institute of Technology: Cambridge, MA. p. 1-4.

29. Saltzer, J.H. and D. Hunt, *Some Recently Repaired Security Holes of Multics*, in *Ancillary Reports: Kernel Design Project*, D.D. Clark, Editor, June 1977, MIT/LCS/TM-87, Laboratory for Computer Science, Massachusetts Institute of Technology: Cambridge, MA. p. 28-33.

30. Saltzer, J.H., P. Jansen, *et al.*, *Some Multics Security Holes Which Were Closed by 6180 Hardware*, in *Ancillary Reports: Kernel Design Project*, D.D. Clark, Editor, June 1977, MIT/LCS/TM-87, Laboratory for Computer Science, Massachusetts Institute of Technology: Cambridge, MA. p. 22-27.

31. Schell, R.R. *Computer security: the Achilles' heel of the electronic Air Force?* in **Air University Review**. January-February 1979, Vol. 30. p. 16-33. URL: http://www.airpower.maxwell.af.mil/airchronicles/aureview/1979/jan-feb/schell.html

32. Schell, R.R., T.F. Tao, *et al. Designing the GEMSOS Security Kernel for Security and Performance.* in **Proceedings of the 8th National Computer Security Conference**. 30 September - 3 October 1985, Gaithersburg, MD DoD Computer Security Center and National Bureau of Standards. p. 108-119.

33. Schiller, W.L., P.T. Withington, *et al.*, *Top Level Specification of a Multics Security Kernel*, WP-20810, 9 July 1976, The MITRE Corporation: Bedford, MA.

34. Schroeder, M.D., *Engineering a Security Kernel for Multics.* **Proceedings of the Fifth Symposium on Operating Systems Principles, Operating Systems Review**, November 1975. **9**(5): p. 25-32.

35. Schroeder, M.D., D.D. Clark, *et al.*, *The Multics Kernel Design Project.* **Proceedings of the Seventh ACM Symposium on Operating Systems Principles, Operating Systems Review**, November 1977. **11**(5): p. 43-56.

36. Schwartz, J. and B. Tedeschi, *New Software Quietly Diverts Sales Commissions.* **The New York Times**, 27 September 2002: p. C1-C4. URL: http://www.nytimes.com/2002/09/27/technology/27FREE.html

37. Smalley, S., *Configuring the SELinux Policy*, NAI Labs Report #02-007, June 2002, NAI Labs: Glenwood, MD. URL: http://www.nsa.gov/selinux/policy2-abs.html

38. Stern, J., *Multics Security Kernel Top Level Specification*, ESD-TR-76-368, November 1976, Honeywell Information Sys-

tems, Inc., McLean, VA, HQ Electronic Systems Division: Hanscom AFB, MA.

39. Thompson, K., *On Trusting Trust.* **Unix Review**, November 1989. **7**(11): p. 70-74.

40. Thompson, K., *Reflections on Trusting Trust.* **Communications of the ACM**, August 1984. **27**(8): p. 761-763.

41. Voydock, V.L., *A Census of Ring 0*, in *Ancillary Reports: Kernel Design Project*, D.D. Clark, Editor, June 1977, MIT/LCS/TM-87, Laboratory for Computer Science, Massachusetts Institute of Technology: Cambridge, MA. p. 5-27.

42. Weissman, C. *BLACKER: Security for the DDN. Examples of A1 Security Engineering Trades.* in **1992 IEEE Computer Society Symposium on Research in Security and Privacy**. 4-6 May 1992, Oakland, CA p. 286-292.

43. Whiteside, T., *Trapdoors and Trojan Horses*, in *Computer Capers*. 1978, Thomas Y. Crowell Co.: New York. p. 115-126.

44. Whitmore, J., A. Bensoussan, *et al.*, *Design for Multics Security Enhancements*, ESD-TR-74-176, December 1973, Honeywell Information Systems, Inc., HQ Electronic Systems Division: Hanscom AFB, MA. URL: http://csrc.nist.gov/publications/history/whit74.pdf

45. Zemmin, F., B. Henk, *et al.*, *Multics Site History: GM*, T.H. Van Vleck, Editor, 20 October 1998. URL: http://www.multicians.org/site-gm.html