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.