# 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

Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC'02)
1063-9527/02 $17.00 © 2002 **IEEE**

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.

1063-9527/02 $17.00 © 2002 IEEE

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.

# 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.

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.

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

# Multics Security Evaluation: Vulnerability Analysis[*]

Paul A. Karger, 2Lt, USAF        Roger R. Schell, Maj, USAF
*Deputy for Command and Management Systems (MCI), HQ Electronic Systems Division*
*Hanscom AFB, MA 01730*

## ABSTRACT

*A security evaluation of Multics for potential use as a two-level (Secret / Top Secret) system in the Air Force Data Services Center (AFDSC) is presented. An overview is provided of the present implementation of the Multics Security controls. The report then details the results of a penetration exercise of Multics on the HIS 645 computer. In addition, preliminary results of a penetration exercise of Multics on the new HIS 6180 computer are presented. The report concludes that Multics as implemented today is not certifiably secure and cannot be used in an open use multi-level system. However, the Multics security design principles are significantly better than other contemporary systems. Thus, Multics as implemented today, can be used in a benign Secret / Top Secret environment. In addition, Multics forms a base from which a certifiably secure open use multi-level system can be developed.*

## 1 INTRODUCTION

### 1.1 Status of Multi-Level Security

A major problem with computing systems in the military today is the lack of effective multi-level security controls. The term multi-level security controls means, in the most general case, those controls needed to process several levels of classified material from unclassified through compartmented top secret in a multi-processing multi-user computer system with simultaneous access to the system by users with differing levels of clearances. The lack of such effective controls in all of today's computer operating systems has led the military to operate computers in a closed environment in which systems are dedicated to the highest level of classified material and all users are required to be cleared to that level. Systems may be changed from level to level, but only after going through very time consuming clearing operations on all devices in the system. Such dedicated systems result in extremely inefficient equipment and manpower utilization and have often resulted in the acquisition of much more hardware than would otherwise be necessary. In addition, many operational requirements cannot be met by dedicated systems because of the lack of information sharing. It has been estimated by the Electronic Systems Division (ESD) sponsored Computer Security Technology Panel [10] that these additional costs may amount to $100,000,000 per year for the Air Force alone.

### 1.2 Requirement for Multics Security Evaluation

This evaluation of the security of the Multics system was performed under Project 6917, Program Element 64708F to meet requirements of the Air Force Data Services Center (AFDSC). AFDSC must provide responsive interactive time-shared computer services to users within the Pentagon at all classification levels from unclassified to top secret. AFDSC in particular did not wish to incur the expense of multiple computer systems nor the expense of encryption devices for remote terminals which would otherwise be processing only unclassified material. In a separate study completed in February 1972, the Information Systems Technology Applications Office, Electronic Systems Division (ESD/MCI) identified the Honeywell Multics system as a candidate to meet both AFDSC's multi-level security requirements and highly responsive advanced interactive time-sharing requirements.

### 1.3 Technical Requirements for Multi-Level Security

The ESD-sponsored Computer Security Technology Planning Study [10] outlined the security weaknesses of present day computer systems and proposed a development plan to provide solutions base on current technology. A brief summary of the findings of the panel follows.

#### 1.3.1 Insecurity of Current Systems

The internal controls of current computers repeatedly have been shown insecure though numerous penetration exercises on such systems as GCOS [9], WWMCCS GCOS [8, 18], and IBM OS/360/370 [16]. This insecurity is a fundamental weakness of contemporary operating sys-

[*] This article is a reprint of a technical report [19] published in June 1974. The program listings from the appendices have been omitted, due to space constraints. The text has been retyped and the figures redrawn, but with no substantive changes. The references have been updated, as some were not yet in final form in 1974.

1063-9527/02 $17.00 © 2002 **IEEE**

tems and cannot be corrected by "patches", "fix-ups", or "add-ons" to those systems. Rather, a fundamental re-implementation using an integrated hardware/software design which considers security as a fundamental requirement is necessary. In particular, steps must be taken to ensure the correctness of the security related portions of the operating system. It is not sufficient to use a team of experts to "test" the security controls of a system. Such a "tiger team" can only show the existence of vulnerabilities but cannot prove their non-existence.

Unfortunately, the managers of successfully penetrated computer systems are very reluctant to permit release of the details of the penetrations. Thus, most reports of penetrations have severe (and often unjustified) distribution restrictions leaving very few documents in the public domain. Concealment of such penetrations does nothing to deter a sophisticated penetrator and can in fact impede technical interchange and delay the development of a proper solution. A system which contains vulnerabilities cannot be protected by keeping those vulnerabilities secret. It can only be protected by the constraining of physical access to the system.

### 1.3.2 Reference Monitor Concept

The ESD Computer Security Technology Panel introduced the concept of a *reference monitor*. This reference monitor is that hardware/software combination which must monitor *all* references by any program to any data anywhere in the system to ensure that the security rules are followed. Three conditions must be met to ensure the security of the system based on a reference monitor.

a. The monitor must be tamper proof.
b. The monitor must be invoked for *every* reference to data anywhere in the system.
c. The monitor must be small enough to be proven correct.

The stated design goals of contemporary systems such as GCOS or OS/360 are to meet the first requirement (albeit unsuccessfully). The second requirement is generally not met by contemporary systems since they usually include "bypasses" to permit special software to operate or must suspend the reference monitor to provide addressability for the operating system in exercising its service functions. The best known of these is the bypass in OS/360 for the IBM supplied service aid, IMASPZAP (SUPERZAP) [2]. Finally and most important, current operating systems are so large, so complex, and so monolithic that one cannot begin to attempt a formal proof or certification of their correct implementation.

### 1.3.3 Hypothesis: Multics is "Securable"

The computer security technology panel identified the general class of descriptor driven processors[1] as extremely useful to the implementation of a reference monitor. Multics, as the most sophisticated of the descriptor-driven systems currently available, was hypothesized to be a potentially securable system; that is, the Multics design was sufficiently well-organized and oriented towards security that the concept of a reference monitor could be implemented for Multics without fundamental changes to the facilities seen by Multics users. In particular, the Multics ring mechanism could protect the monitor from malicious or inadvertent tampering, and the Multics segmentation could enforce monitor mediation on *every* reference to data. However, the question of certifiability had not as yet been addressed in Multics. Therefore the Multics vulnerability analysis described herein was undertaken to:

a. Examine Multics for potential vulnerabilities.
b. Identify whether a reference monitor was practical for Multics.
c. Identify potential interim enhancements to Multics to provide security in a benign (restricted access) environment.
d. Determine the scope and dimension of a certification effort.

## 1.4 Sites Used

The vulnerability analysis described herein was carried out on the HIS 645 Multics Systems installed at the Massachusetts Institute of Technology and at the Rome Air Development Center. As the HIS 6180, the new Multics processor, was not available at the time of the study, this report will describe results of analysis of the HIS 645 only. Since the completion of the analysis, work has started on an evaluation of the security controls of Multics on the HIS 6180. Preliminary results of the work on the HIS 6180 are very briefly summarized in this report, to provide an understanding of the value of the evaluation of the HIS 645 in the context of the new hardware environment.

# 2 MULTICS SECURITY CONTROLS

This section provides a brief overview of the basic Multics security controls to provide necessary background for the discussion of the vulnerability analysis. However, a rather thorough knowledge of the Multics implementation is assumed throughout the rest of this document. More complete background material may be found in Lipner [21], Saltzer [25], Organick [22], and the **Multics Programmers' Manual** [4].

The basic security controls of Multics fall into three major areas: hardware controls, software controls, and procedural controls. This overview will touch briefly on each of these areas.

[1] Descriptor driven processors use some form of address translation though hardware interpretation of descriptor words or registers. Such systems include the Burroughs 6700, the Digital Equipment Corp. PDP-11/45, the Data General Nova 840, the DEC KI-10, the HIS 6180, the IBM 370/158 and 168, and several others not listed here.

## 2.1 Hardware Security Controls

### 2.1.1 Segmentation Hardware

The most fundamental security controls in the HIS 645 Multics are found in the segmentation hardware. The basic instructions set of the 645 can directly address up to 256K[2] distinct segments[3] at any one time, each segment being up to 256K words long.[4] Segments are broken up into 1K word pages[5] which can be moved between primary and secondary storage by software, creating a very large virtual memory. However, we will not treat paging throughout most of this evaluation as it is transparent to security. Paging must be implemented correctly in a secure system. However, bugs in page control are generally difficult to exploit in a penetration, because the user has little or no control over paging operations.

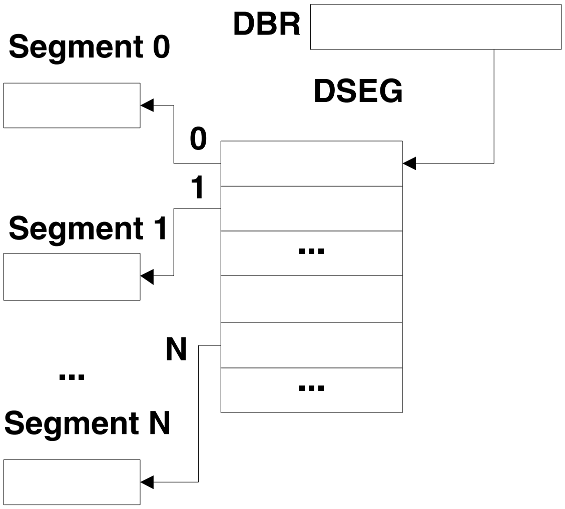


**Figure 1. Segmentation Hardware**

Segments are accessed by the 645 CPU through segment descriptor words (SDW's) that are stored in the descriptor segment (DSEG). (See Figure 1.) To access segment N, the 645 CPU uses a processor register, the descriptor segment base register (DBR), to find the DSEG. It then accesses the Nth SDW in the DSEG to obtain the address of the segment and the access rights currently in force on that segment for the current user.

Each SDW contains the absolute address of the page table for the segment and the access control information. (See Figure 2.) The last 6 bits of the SDW determine the access rights to the segment - read, execute, write, etc.[6] Using these access control bits, the supervisor can protect the descriptor segment from unauthorized modification by denying access in the SDW for the descriptor segment.

| 0 17 | 18 29 | 30 | 31 | 32 | 33 35 |
|---|---|---|---|---|---|
| ADDR | OTHER | WRITE PERMIT | SLAVE ACC. | OTHER | CLASS |

**Meaning of CLASS field**
**0 = FAULT**
**1 = DATA**
**2 = SLAVE PROCEDURE**
**3 = EXECUTE ONLY**
**4 = MASTER PROCEDURE**
**5, 6, 7 = ILLEGAL DESCRIPTOR**

**Figure 2. SDW Format**

### 2.1.2 Master Mode

To protect against unauthorized modification of the DBR, the processor operates in one of two states – master mode and slave mode. In master mode, any instruction may be executed and access control checks are inhibited.[7] In slave mode, certain instructions, including those which modify the DBR, are inhibited. Master mode procedure segments are controlled by the class field in the SDW. Slave mode procedures may transfer to master mode procedures *only* through word zero of the master mode procedure to prevent unrestricted invocation of privileged programs. It is then the responsibility of the master mode software to protect itself from malicious calls by placing suitable protective routines beginning at location zero.

## 2.2 Software Security Controls

The most outstanding feature of the Multics security controls is that they operate on basis of "form" rather than the classical basis of "content". That is to say, the Multics controls are based on operations on a uniform population of well defined objects, as opposed to the classical controls which rely on anticipating all possible types of accesses and make security essentially a battle of wits.

### 2.2.1 Protection Rings

The primary software security control on the 645 Multics system is the ring mechanism. It was originally postulated as desirable to extend the traditional master/slave mode relationship of conventional machines to permit layering within the supervisor and within user code (see Gra-

[2] 1K = 1024 units.

[3] Current software table sizes restrict a process to about 1000 segments. However, by increasing these table sizes, the full hardware potential may be used.

[4] The 645 software restricted segments to 64K words for efficiency reasons.

[5] The 645 hardware also support 64 word pages which were not used. The 6180 supports only a single page size which can be varied by field modification from 64 words to 4096 words. Initially, a size of 1024 words is being used. The supervisors on both the 645 and 6180 use unpaged segments of length 0 mod 64.

[6] A more detailed description of the SDW format may be found in the 645 processor manual [11].

[7] The counterpart of master mode on the HIS 6180, called privileged mode, does not inhibit access control checking.

1063-9527/02 $17.00 © 2002 IEEE

ham [17]). Eight concentric rings of protection, numbered 0 – 7, are defined with higher numbered rings having less privilege than lower numbered rings, and with ring 0 containing the *hardcore* supervisor.[8] Unfortunately, the 645 CPU does not implement protection rings in hardware.[9] Therefore, the eight protection rings are implemented by providing eight descriptor segments for each process (user), one descriptor segment per ring. Special fault codes are placed in those SDW's which can be used for cross-ring transfers so that ring 0 software can intervene and accomplish the descriptor segment swap between the calling and called rings.

### 2.2.2 Access Control Lists

Segments in Multics are stored in a hierarchy of directories. A directory is a special type of segment that is not directly accessible to the user and provides a place to store names and other information about subordinate segments and directories. Each segment and directory has an access control list (ACL) in its parent directory entry controlling who may *r*ead (r), *w*rite (w), or *e*xecute (e) the segment or obtain *s*tatus (s) of, *m*odify (m) entries in, or *a*ppend (a) entries to a directory. For example in Figure 3, the user Jones.Druid has read permission to segment ALPHA and has null access to segment BETA. However, Jones.Druid has modify permission to directory DELTA, so he can give himself access to BETA. Jones.Druid cannot give himself write access to segment ALPHA, because he does not have modify permission to directory GAMMA. In turn, the right to modify the access control lists of GAMMA and DELTA is controlled by the access control list of directory EPSILON, stored in the parent of EPSILON. Access control security checks for segments are enforced by the ring 0 software by setting the appropriate bits in the SDW at the time that a user attempts to add a segment to his address space.

### 2.2.3 Protected Access Identification

In order to do access checking, the ring 0 software must have a protected, non-forgeable identification of a user to compare with the ACL entries. This ID is established when a user signs on to Multics and is stored in the process data segment (PDS) which is accessible only in ring 0 or in master mode, so that the user may not tamper with the data stored in the PDS.

### 2.2.4 Master Mode Conventions

By convention, to protect master mode software, the original design specified that master mode procedures were not to be used outside ring 0. If the master mode procedure ran in the user ring, the master mode procedure itself would be forced to play the endless game of wits of the classical supervisor call. The master mode procedure would have to include code to check for all possible combinations of input arguments, rather than relying on a fundamental set of argument independent security controls. As an aid (or perhaps hindrance) to playing the game of wits, each master mode procedure must have a master mode pseudo-operation code assembled into location 0. The master mode pseudo-operation generates code to test an index register for a value corresponding to an entry point in the segment. If the index register is invalid, the master-mode pseudo-operation code saves the registers for debugging and brings the system down.

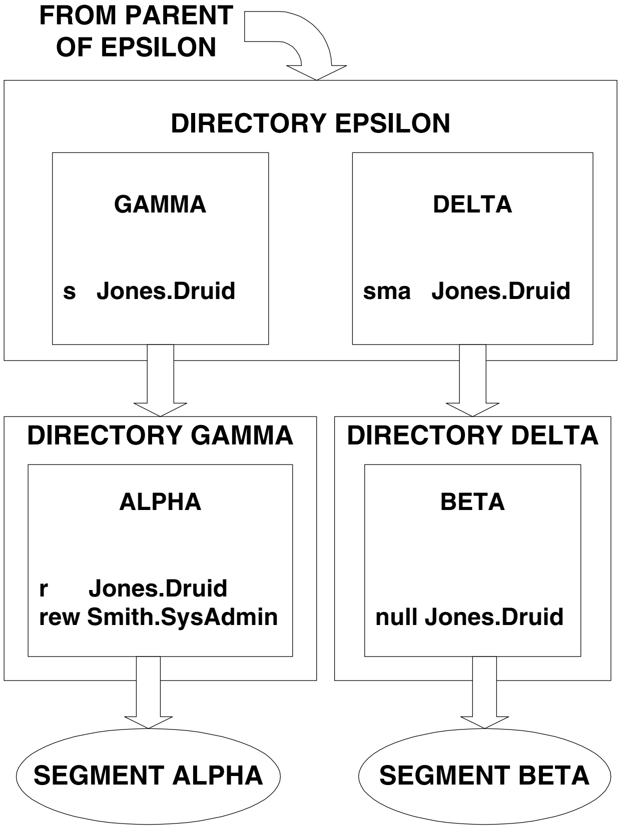


**Figure 3. Directory Hierarchy**

## 2.3 Procedural Security Controls

### 2.3.1 Enciphered Passwords

When a user logs in to Multics, he types a password as his primary authentication. Of course, the access control list of the password file denies access to regular users of the system. In addition, as a protection against loss of a system dump which could contain the password file, all passwords are stored in a *non-invertible* cipher form. When a user types his password, it is enciphered and com-

[8] The original design called for 64 rings, but this was reduced to 8 in 1971.

[9] One of the primary enhancements of the HIS 6180 is the addition of ring hardware [28] and a consequent elimination of the need for master mode procedures in the user ring.

pared with the stored enciphered version for validity. Clear text passwords are stored nowhere in the system.

### 2.3.2 Login Audit Trail

Each login and logout is carefully audited to check for attempts to guess valid user passwords. In addition, each user is informed of the date, time and terminal identification (if any) of the last login to detect past compromises of the user's access rights. Further, the user is told the number of times his password has been given incorrectly since its last correct use.

### 2.3.3 Software Maintenance Procedures

The maintenance of the Multics software is carried out online on a dial-up Multics facility. A systems programmer prepares and nominally debugs his software for installation. He then submits his software to a library installer who copies and recompiles the source in a protected directory. The library installer then checks out the new software prior to installing it in the system source and object libraries. Ring 0 software is stored on a system tape that is reloaded into the system each time it is brought up. However, new system tapes are generated from online copies of the ring 0 software. The system libraries are protected against modification by the standard ACL mechanism. In addition, the library installers periodically check the date/time last modified of all segments in the library in an attempt to detect unauthorized modifications.

# 3 VULNERABLITY ANALYSIS

## 3.1 Approach Plan

It was hypothesized that although the fundamental design characteristics of Multics were sound, the implementation was carried out on an ad hoc basis and had security weaknesses in each of the three areas of security controls described in Section 2 – hardware, software, and procedures.

The analysis was to be carried out on a very limited basis with less than one-half man month per month level of effort. Due to the manpower restrictions, a goal of one vulnerability per security control area was set. The procedure followed was to postulate a weakness in a general area, verify the weakness in the system, experiment with the weakness on the Rome Air Development Center (RADC) installation, and finally, using the resulting debugged penetration approach, exploit the weakness on the MIT installation.

An attempt was to be made to operate with the same type of ground rules under which a real agent would operate. That is, with each penetration, an attempt would be made to extract or modify sensitive system data without detection by the system maintenance or administrative personnel.

Several exploitations were successfully investigated. These included changing access fields in SDW's, changing protected identities in the PDS, inserting trap doors into the system libraries, and accessing the system password file.

## 3.2 Hardware Vulnerabilities

### 3.2.1 Random Failures

One area of significant concern in a system processing multi-level classified material is that of random hardware failures. As described in Section 2.1.1, the fundamental security of the system is dependent on the correct operation of the segmentation hardware. If this hardware is prone to error, potential security vulnerabilities become a significant problem.

To attempt a gross measure of the rate of security sensitive component failure, a procedure called the *subverter* was written to sample the security sensitive hardware on a frequent basis, testing for component failures which could compromise the security controls. The subverter was run in the background of an interactive process. Once each minute, the subverter received a timer interrupt and performed one test from the list described below. Assuming the test did not successfully violate security rules, the sub-

| | 1100 operating hours | |
|---|---|---|
| | Test Name | Number Attempts |
| 1. | Clear Associative Memory | 3526 |
| 2. | Store Control Unit | 3466 |
| 3. | Load Timer Unit | 3444 |
| 4. | Load Descriptor Base Register | 3422 |
| 5. | Store Descriptor Base Register | 3403 |
| 6. | Connect I/O Channel | 3378 |
| 7. | Delay Until Interrupt Signal | 3359 |
| 8. | Read Memory Controller Mask Register | 3344 |
| 9. | Set Memory Controller Mask Register | 3328 |
| 10. | Set Memory Controller Interrupt Cells | 3309 |
| 11. | Load Alarm Clock | 3289 |
| 12. | Load Associative Memory | 3259 |
| 13. | Store Associative Memory | 3236 |
| 14. | Restore Control Unit | 3219 |
| 15. | No Read Permission | 3148 |
| 16. | No Write Permission | 3131 |
| 17. | XED – No Read Permission | 3113 |
| 18. | XED – No Write Permission | 3098 |
| 19. | Tally Word Without Write Permission | 3083 |
| 20. | Bounds Fault <64K | 2398 |
| 21. | Bounds Fault >64K | 2368 |
| 22. | Illegal Opcodes | 2108 |

**Table 1. Subverter Test Attempts**

verter would go to sleep for one minute before trying the next test. A listing of the subverter may be found in Appendix A.

The subverter was run for 1100 hours in a one year period on the MIT 645 system. The number of times each test was attempted is shown in Table 1. During the 1100 operating hours, no security sensitive hardware component failures were detected, indicating good reliability for the 645 security hardware. However, two interesting anomalies were discovered in the tests. First, one undocumented instruction (octal 471) was discovered on the 645. Experimentation indicated that the new instruction had no obvious impact on security, but merely seemed to store some internal register of no particular interest. The second anomaly was a design error resulting in an algorithmic failure of the hardware described in Section 3.2.2.

Tests 1-14 are tests of master mode instructions. Tests 15 and 16 attempt simple violation of read and write permission as set on segment ACL's. Tests 17 and 18 are identical to 15 and 16 except that the faulting instructions are reached from an Execute Double instruction rather than normal instruction flow. Test 19 attempts to increment a tally word that is in a segment without write permission. Tests 20 and 21 take out of bounds faults on segments of zero length, forcing the supervisor to grow new page tables for them. Test 22 attempts execution of all the instructions marked illegal on the 645.

### 3.2.2 Execute Instruction Access Check Bypass

While experimenting with the hardware subverter, a sequence of code[10] was observed which would cause the hardware of the 645 to bypass access checking. Specifically, the execute instruction in certain cases described below would permit the executed instruction to access a segment for reading or writing without the corresponding permissions in the SDW.

This vulnerability occurred when the execute instruction was in certain restricted locations of a segment with at least read-execute (re) permission. (See Figure 4.) The execute instruction then referenced an object instruction in word zero of a second segment with at least R permission. The object instruction indirected through an ITS pointer in the first segment to access a word for reading or writing in a third segment. The third segment was required to be *active*; that is, to have an SDW pointing to a valid page table for the segment. If all these conditions were met *precisely*, the access control fields in the SDW of the third segment would be ignored and the object instruction permitted to complete without access checks.

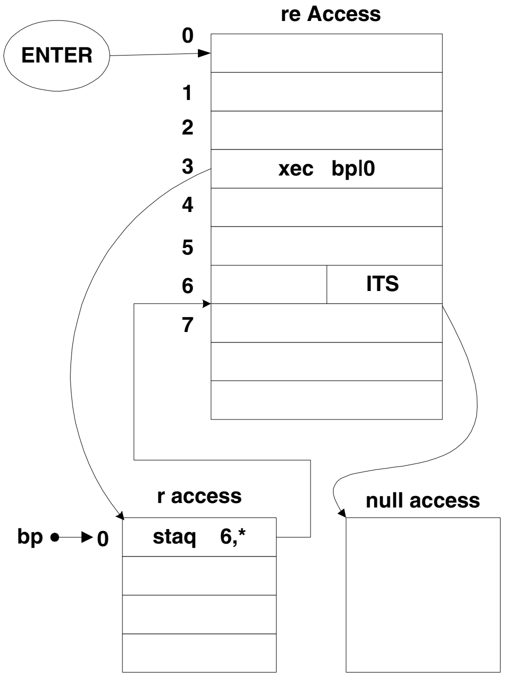


**Figure 4. Execute Instruction Bypass**

The exact layout of instructions and indirect words was crucial. For example, if the object instruction used a base register rather than indirecting through the segment containing the execute instruction (i.e., staq apl0 rather than staq 6,*), then the access checks were done properly. Unfortunately, a complete schematic of the 645 was not available to determine the exact cause of the bypass. In informal communications with Honeywell, it was indicated that the error was introduced in a field modification to the 645 at MIT and was then made to all processors at all other sites.

This hardware bug represents a violation of one of the most fundamental rules of the Multics design - the checking of *every* reference to a segment by the hardware. This bug was not caused by fundamental design problems. Rather, it was caused by carelessness by the hardware engineering personnel.

No attempt was made to make a complete search for additional hardware design bugs, as this would have required logic diagrams for the 645. It was sufficient for this effort to demonstrate one vulnerability in this area.

### 3.2.3 Preview of 6180 Hardware Vulnerabilities

While no detailed look has been taken at the issue of hardware vulnerabilities on the 6180, the very first login of an ESD analyst to the 6180 inadvertently discovered a hardware vulnerability that crashed the system. The vulnerability was found in the Tally Word Without Write Permission test of the subverter. In this test, when the 6180 processor encountered the tally word without write permission, it signalled a *trouble* fault rather than an *access violation* fault. The *trouble* fault is normally signalled only when a fault occurs during the signalling of a fault. Upon

[10] The subverter was designed to test sequences of code in which single failures could lead to security problems. Some of these sequences exercised relatively complex and infrequently used instruction modifications, which experience had shown were prone to error.

encountering a *trouble* fault, the software normally brings the system down.

It should be noted that the HIS 6180 contains very new and complex hardware that, as of this publication, has not been completely "shaken down". Thus, Honeywell still quite reasonably expects to find hardware problems. However, the inadequacy of "testing" for security vulnerabilities applies equally well to hardware as to software. Simply "shaking down" the hardware cannot find all the possible vulnerabilities.

## 3.3 Software Vulnerabilities

Although the approach plan for the vulnerability analysis only called for locating one example of each class of vulnerability, three software vulnerabilities were identified as shown below. Again, the search was neither exhaustive nor systematic.

### 3.3.1 Insufficient Argument Validation

Because the 645 Multics system must simulate protection rings in software, there is no direct hardware validation of arguments passed in a subroutine call from a less privileged ring to a more privileged ring. Some form of validation is required, because a malicious user could call a ring 0 routine that stores information through a user supplied pointer. If the malicious user supplied a pointer to data to which ring 0 had write permission but to which the user ring did not, ring 0 could be *tricked* into causing a security violation.

To provide validation, the 645 software ring crossing mechanism requires all gate segments[11] to declare to the *gatekeeper* the following information.

1. number of arguments expected
2. data type of each argument
3. access requirements for each argument - read only or read/write

This information is stored by convention in specified locations within the gate segment.[12] The *gatekeeper* invokes an argument validation routine that inspects the argument list being passed to the gate to ensure that the declared requirements are met. If any test fails, the argument validator aborts the call and signals the condition *gate error* in the calling ring.

In February 1973, a vulnerability was identified in the argument validator that would permit the "fooling" of ring 0 programs. The argument validator's algorithm to validate read or read/write permission was as follows: First copy the argument list into ring 0 to prevent modification of the argument list by a process running on another CPU in the system while the first process is in ring 0 and has completed argument validation. Next, force indirection through each argument pointer to obtain the segment number of the target argument. Then look up the segment in the calling ring's descriptor segment to check for read or write permission.

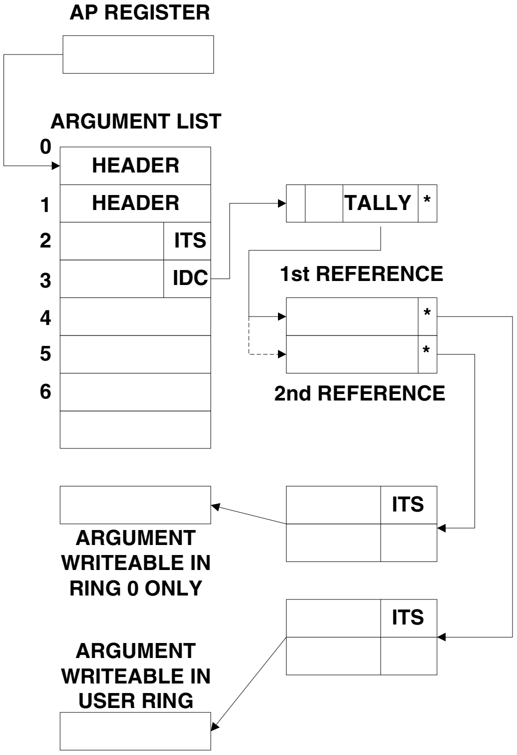


**Figure 5. Insufficient Argument Validation**

The vulnerability is as follows: (See figure 5.) An argument pointer supplied by the user is constructed to contain an IDC modifier (*i*ncrement address, *d*ecrement tally, and *c*ontinue) that causes the first reference though the indirect chain to address a valid argument. This first reference is the one made by the argument validator. The reference through the IDC modifier increments the address field of the tally word causing it to point to a different indirect word which in turn points to a different ITS pointer which points to an argument which is writable in ring 0 only. The second reference through this modified indirect chain is made by the ring 0 program, which proceeds to write data where it shouldn't.[13]

This vulnerability resulted from violation of a basic rule of the Multics design – that all arguments to a more privileged ring be validated. The problem was not in the fun-

[11] A gate segment is a segment used to cross rings. It is identified by R2 and R3 of its ring brackets R1, R2, R3 being different. See Organick [22] for a detailed description of ring brackets.

[12] For the convenience of authors of gates, a special "gate language" and "gate complier" are provided to generate properly formatted gates. Using this language, the author of the gate can declare the data type and access requirement of each argument.

[13] Depending on the actual number of references made, the malicious user need only vary the number of indirect words pointing to legal and illegal arguments. We have assumed for simplicity here that the validator and the ring 0 program make only one reference each.

damental design – the concept of a software argument validator is sound given the lack of ring hardware. The problem was an ad hoc implementation of that argument validator which overlooked a class of argument pointers.

Independently, a change was made to the MIT system, which fixed this vulnerability in February 1973. The presence and exploitability of the vulnerability were verified on the RADC Multics, which had not been updated to the version running at MIT. The method of correction chosen by MIT was rather *brute force*. The argument validator was changed to require the modifier in the second word of each argument pointer always to be zero. This requirement solves the specific problem of the IDC modifier, but not the general problem of argument validation.

### 3.3.2 Master Mode Transfer

As described in Sections 2.1.2 and 2.2.4, the 645 CPU has a master mode in which privileged instructions may be executed and in which access checking is inhibited although address translation through segment and page tables is retained.[14] The original design of the Multics protection rings called for master mode code to be restricted to ring 0 by convention.[15] This convention caused the fault handling mechanism to be excessively expensive due to the necessity of switching from the user ring into ring 0 and out again using the full software ring crossing mechanism. It was therefore proposed and implemented that the *signaller*, the module responsible for processing faults to be signalled to the user,[16] be permitted to run in the user ring to speed up fault processing. The signaller is a master mode procedure, because it must execute the RCU (*R*estore *C*ontrol *U*nit) instruction to restart a process after a fault.

The decision to move the signaller to the user ring was not felt to be a security problem by the system designers, because master mode procedures could only be entered at word zero. The signaller would be assembled with the master mode pseudo-operation code at word zero to protect it from any malicious attempt by a user to execute an arbitrary sequence of instructions within the procedure. It was also proposed, although never implemented, that the code of master mode procedures in the user ring be specially audited. However as we shall see in Section 3.4.4, auditing does not guarantee victory in the *battle of wits* between the implementer and the penetrator. Auditing cannot be used to make up for fundamental security weaknesses.

It was postulated in the ESD/MCI vulnerability analysis that master mode procedures in the user ring represent a fundamental violation of the Multics security concept. Violating this concept moves the security controls from the basic hardware/software mechanism to the cleverness of the systems programmer who, being human, makes mistakes and commits oversights. The master mode procedures become classical *supervisor calls* with no rules for *sufficient* security checks. In fact, upon close examination of the signaller, this hypothesis was found to be true.

The master mode pseudo-operation code was designed only to protect master mode procedures from random calls with ring 0. It was not designed to withstand the attack of a malicious user, but only to operate in the relatively benign environment of ring 0.

```
        name          master_test
        mastermode
        entry         a
        entry         b
 a:     code
        ...
 b:     code
        ...
        end
```

**Figure 6. Master Mode Source Code**

The master mode program shown in Figure 6 assembles into the interpreted object code shown in Figure 7. The master mode procedure can only be entered at location zero.[17] By convention, the n entry points to the procedure

```
    cmpx0  2,du      "call in bounds?
    tnc    transfer_vector,0
                     "Yes, go to entry
    stb    sp|0      "Illegal call here
    sreg   sp|10     "save registers
    stcd   sp|24
    tra    lp|12,*   "lp|12 points
                     "to mxerror
a:  code
    ...
b:  code
    ...
transfer_vector:
    tra    a
    tra    b
    end
```

**Figure 7. Master Mode Interpreted Object Code**

are numbered from 0 to n-1. The number of the desired entry point must be in index register zero at the time of the call. The first two instructions in the master mode sequence check to ensure that index register zero is in bounds. If it is, the transfer on no carry (`tnc`) instruction indirects through the transfer vector to the proper entry. If

[14] The 645 also has an absolute mode in which all addresses are absolute core addresses rather than being translated by the segmentation hardware. This mode is used only to initialize the system.

[15] This convention is enforced on the 6180. Privileged mode (the 6180 analogy to the 645 master mode) only has effect in ring 0. Outside ring 0, the hardware ignores the privileged mode bit.

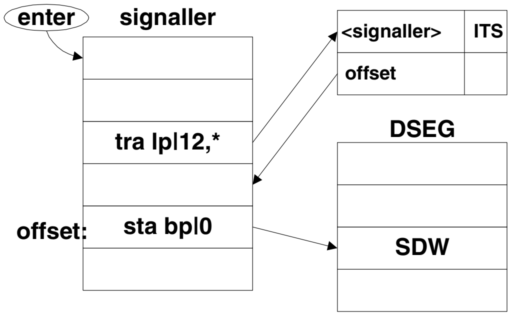
[16] The signaller processed such faults as zerodivide and access violation, which are signalled to the user. Page faults and segment faults, which the user never sees, are processed elsewhere in ring 0.

[17] This restriction is enforced by hardware described in Section 2.1.2.

index register zero is out of bounds, the processor registers are saved for debugging and control is transferred to *mxerror*, a routine to crash the system because of an unrecoverable error.

The transfer to *mxerror* is the most obvious vulnerability. By moving the signaller into the user ring, the designers allowed a user to arbitrarily crash the system by transferring to `signaller|0` with a bad value in index register zero. This vulnerability is not too serious, since it does not compromise information and could be repaired by changing *mxerror* to handle the error, rather than crashing the system.

However, there is a much more subtle and dangerous vulnerability here. The `tra lp|12,*` instruction that is used to call *mxerror* believes that the lp register points to the linkage section of the signaller, which it should if the call were legitimate. However, a malicious user may set the lp register to point wherever he wishes, *permitting him to transfer to an arbitrary location while the CPU is still in master mode.* The key is the transfer in master mode, because this permits a transfer to an arbitrary location within another master mode procedure without access checking and without the restriction of entering at word zero. Thus, the penetrator need only find a convenient store instruction to be able to write into his own descriptor segment, for example. Figure 8 shows the use of a `sta bp|0` instruction to change the contents of an SDW illegally.



**Setup Conditions**

```
A reg :=  new SDW
Index 0 :=  -1
lp := address(POINTER)-12
POINTER := address(sta instruction)
bp := address(SDW)
```

**Figure 8. Store with Master Mode Transfer**

There is one major difficulty in exploiting this vulnerability. The instruction to which control is transferred must be chosen with extreme care. The instructions immediately following the store must provide some orderly means of returning control to the malicious user without doing uncontrolled damaged to the system. If a crucial data base is garbled, the system will crash leaving a core dump which could incriminate the penetrator.

This vulnerability was identified by ESD/MCI in June 1972. An attempt to use the vulnerability led to a system crash for the following reason: Due to an obsolete listing of the signaller, the transfer was made to an ldbr (*L*oad *D*escriptor *B*ase *R*egister) instruction instead of the expected store instruction. The DBR was loaded with a garbled value, and the system promptly crashed. The system maintenance personnel, being unaware of the presence of an active penetration, attributed the crash to a disk read error.

The Master Mode Transfer vulnerability resulted from a violation of the fundamental rule that master mode code shall not be executed outside ring 0. The violation was not made maliciously by the system implementers. Rather it occurs because of the interaction of two seemingly independent events: the ability to transfer via the lp without the system being able to check the validity of the lp setting, and the ability for that transfer to be master mode code. The separation of these events made the recognition of the problem unlikely during implementation.

### 3.3.3 Unlocked Stack Base

The 645 CPU has eight 18-bit registers that are used for inter-segment references. Control bits are associated with each register to allow it to be paired with another register as a word number-segment number pair. In addition, each register has a lock bit, settable only in master mode, which protects its contents from modification. By convention, the eight registers are named and paired as shown in Table 2.

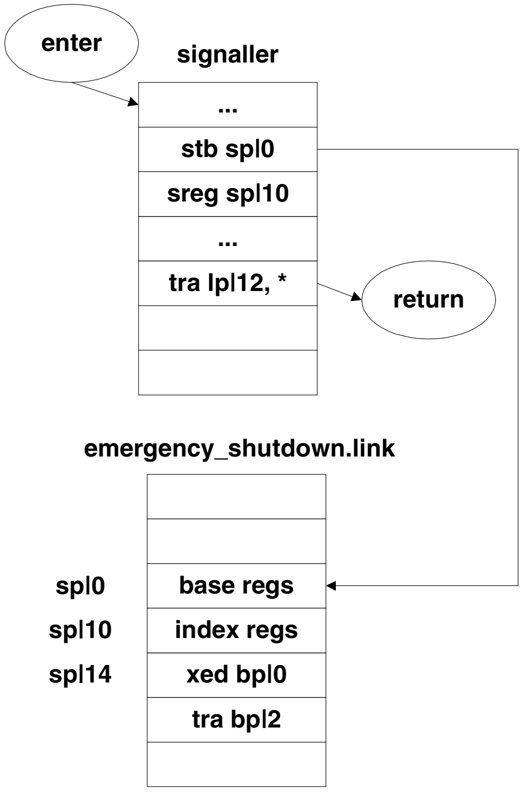
| Num | Name | Use | Pairing |
|---|---|---|---|
| 0 | ap | Argument pointer | Paired with ab |
| 1 | ab | Argument base | Unpaired |
| 2 | bp | Unassigned | Paired with bb |
| 3 | bb | Unassigned | Unpaired |
| 4 | lp | Linkage pointer | Paired with lb |
| 5 | lb | Linkage base | Unpaired |
| 6 | sp | Stack pointer | Paired with sb |
| 7 | sb | Stack base | unpaired |

**Table 2. Base Register Pairing**

During the early design of the Multics operating system, it was felt that the ring 0 code could be simplified if the stack base (sb) register were locked, that is, could only be modified in master mode. The sb contained the segment number of the user stack which was guaranteed to be writeable. If the sb were locked, then the ring 0 fault and interrupt handlers could have convenient areas in which to store stack frames. After Multics had been released to users at MIT, it was realized that locking the stack base unnecessarily constrained language designers. Some languages would be extremely difficult to implement without the capability of quickly and easily switching between stack segments. Therefore, the system was modified to no longer lock the stack base.

When the stack base was unlocked, it was realized that there was code scattered throughout ring 0 which assumed that the sb always pointed to the stack. Therefore, ring 0 was *audited* for all code which depended on the locked stack base. However, the audit was never completed and the few dependencies identified were in general not repaired until much later.

As part of the vulnerability analysis, it was hypothesized that such an audit for unlocked stack base problems was presumably incomplete. The ring 0 code is so large that a subtle dependency on the sb register could easily slip by an auditor's notice. This, in fact proved to be true as shown below:



**Setup Conditions**

**AQ register := xed bp|0; tra bp|2**
**Index 0 := -1**
**sp := address(unused storage in**
**emergency_shutdown.link)**
**lp|12 := address(return location)**

**Figure 9. Unlocked Stack Base (Step 1)**

Section 3.3.2 showed that the master mode pseudo-operation code believed the value in the lp register and transferred through it. Figure 7 shows that the master mode pseudo-operation code also depends on the sb pointing to a writeable stack segment. When an illegal master mode call is made, the registers are saved on the stack prior to calling *mxerror* to crash the system. This code was designed prior to the unlocking of the stack base and was not detected in the system audit. The malicious user need only set the sp-sb pair to point anywhere to perform an illegal store of the registers with master mode privileges.

The exploitation of the unlocked stack base vulnerability was a two step procedure. The master mode pseudo-operation code stored *all* the processor registers in an area over 20 words long. This area was far too large for use in a system penetration in which at most one or two words are modified to give the agent the privileges he requires. However, storing a large number of words could be very useful to install a *trap door* in the system -- that is a sequence of code which when properly invoked provides the penetrator with the needed tools to subvert the system. Such a *trap door* must be well hidden to avoid accidental discovery by the system maintenance personnel.

It was noted that the linkage segments of several of the ring 0 master mode procedures were preserved as separate segments rather than being combined in a single linkage segment. Further, these linkage segments were themselves master mode procedures. Thus, segments such as signaller, fim, and emergency_shutdown had corresponding master mode linkage segments signaller.link, fim.link, and emergency_shutdown.link. Linkage segments contain a great deal of information used only by the binder and therefore contain a great deal of extraneous information in ring 0. For this reason, a master mode linkage segment is an ideal place to conceal a *trap door*. There is a master mode procedure called emergency_shutdown that is used to place the system in a consistent state in the event of a crash. Since emergency_shutdown is used only at the time of a system crash, its linkage segment, emergency_shutdown.link, was chosen to be used for the *trap door*.

The first step of the exploitation of the unlocked stack base is shown in Figure 9.[18] The signaller is entered at location 0 with an invalid index register 0. The stack pointer is set to point to an area of extraneous storage in emergency_shutdown.link. The AQ register contains a two instruction *trap door* which when executed in master mode can load or store any 36-bit word in the system. The index registers could be used to hold a longer *trap door*; however, in this case the xed bp|0, tra bp|2 sequence is sufficient. The base registers, index registers, and AQ register are stored into emergency_shutdown.link, thus laying the *trap door*. Finally a transfer is made indirect through lp|12 which has been pre-set as a return pointer.[19]

Step two of the exploitation of the unlocked stack base is shown in Figure 10. The calling program sets the bp register to point to the desired instruction pair and transfers to word zero of the signaller with an invalid value in index

[18] Listings of the code used to exploit this vulnerability are found in Appendix B.

[19] This transfer uses the Master Mode Transfer vulnerability to return. This is done primarily for convenience. The fundamental vulnerability is the storing though the sp register. Without the Master Mode Transfer, exploitation of the Unlocked Stack Base would have been more difficult, although far from impossible.

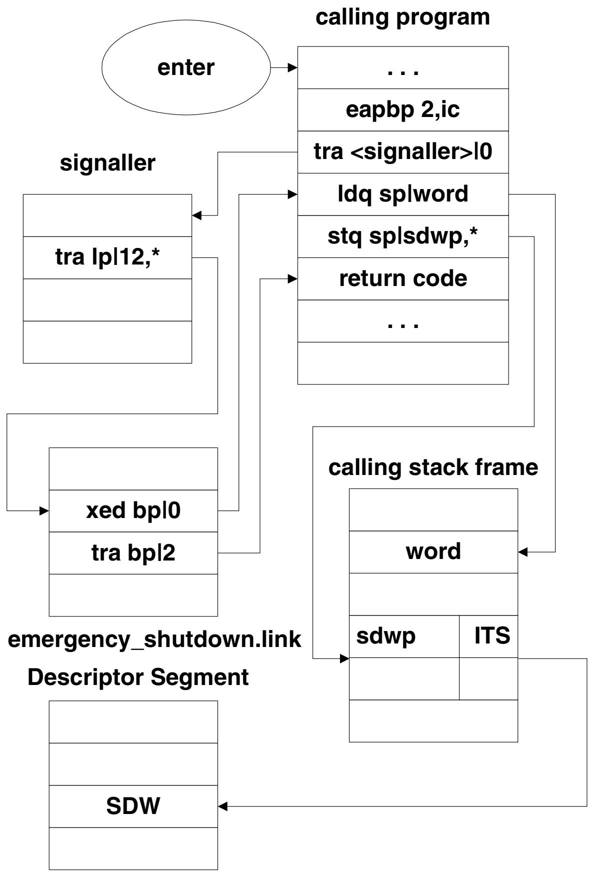


**Figure 10. Unlocked Stack Base (Step 2)**

register 0. The signaller saves its registers on the user's stack frame since the sp has not been changed. It then transfers indirect through lp|12 which has been set to point to the *trap door* in emergency_shutdown.link. The first instruction of the *trap door* is an execute double (XED) which permits the user (penetration agent) to specify any two arbitrary instructions to be executed *in master mode*. In this example, the instruction pair loads the Q register from a word in the stack frame[20] and then stores indirect through a pointer in the stack to an SDW in the descriptor segment. The second instruction in the *trap door* transfers back to the calling program, and the penetrator may go about his business.

The *trap door* inserted in emergency_shutdown.link remained in the system until the system was reinitialized.[21] At initialization time, a fresh copy of all ring zero segments is read in from the system tape erasing the *trap door*. Since system initializations occur at least once per day, the penetrator must execute step one before each of his working sessions. Step two is then executed each time he wishes to access or modify some word in the system.

The unlocked stack base vulnerability was identified in June 1972 with the Master Mode Transfer Vulnerability. It was developed and used at the RADC site in September 1972 without a single system crash. In October 1972, the code was transferred to the MIT site. Due to lack of good telecommunications between the two sites, the code was manually retyped into the MIT system. A typing mistake was made that caused the word to be stored into the SDW to always be zero (See Figure 10). When an attempt was made to set slave access-data in the SDW of the descriptor segment itself,[22] the SDW of the descriptor segment was set to zero causing the system to crash at the next LDBR instruction or segment initiation. The bug was recognized and corrected immediately, but later in the day, a second crash occurred when the SDW for the ring zero segment fim (the *f*ault *i*ntercept *m*odule) was patched to slave access-write permit-data rather than slave access-write permit-slave procedure. In more straightforward terms, the SDW was set to read-write rather than read-write-execute. Therefore, when the system next attempted to execute the fim it took a no-execute permission fault and tried to execute the fim, thus entering an infinite loop crashing the system.

### 3.3.4 Preview of 6180 Software Vulnerabilities

The 6180 hardware implementation of rings renders invalid the attacks described here on the 645. This is not to say, however, that the 6180 Multics is free of vulnerabilities. A cursory examination of the 6180 software has revealed the existence of several software vulnerabilities, any one of which can be used to access any information in the system. These vulnerabilities were identified with comparable levels of effort to those shown in Section 3.5.

#### 3.3.4.1 No Call Limiter Vulnerability

The first vulnerability is the No Call Limiter vulnerability. This vulnerability was caused by the call limiter not being set on gate segments, allowing the user to transfer to any instruction within the gate rather than to just an entry transfer vector. This vulnerability gives the penetrator the same capabilities as the Master Mode Transfer vulnerability.

#### 3.3.4.2 SLT-KST Dual SDW Vulnerability

The second vulnerability is the SLT-KST Dual SDW vulnerability. When a user process was created on the 645, separate descriptor segments were created for each ring with the ring 0 SDW's being copied from the segment loading table (SLT). The ring 0 descriptor segment was

[20] It should be noted that only step one changed the value of the sp. In step two, it is very useful to leave the sp pointing to a valid stack frame.
[21] See Section 3.4.5 for more lasting "trap doors".

[22] The attempt here was to dump the contents of the descriptor segment on the terminal. The user does not normally have read permission to his own descriptor segment.

essentially a copy of the SLT for ring 0 segments. The ring 4 descriptor segment zeroed out most SDW's for ring 0 segments. Non-ring 0 SDW's were added to both the ring 0 and ring 4 descriptor segments from the Known Segment Table (KST) during segment initiation. Upon conversion to the 6180, the separate descriptor segments for each ring were merged into one descriptor segment containing ring brackets in each SDW [5]. The ring 0 SDW's were still taken from the SLT and the non-ring 0 SDW's from the KST as on the 645.

The system contains several gates from ring 4 into ring 0 of varying levels of privilege. The least privileged gate is called hcs_ and may be used by all users in ring 4. The most privileged gate is called hphcs_ and may only be called by system administration personnel. The gate hphcs_ contains routines to shut the system down, access any segment in the system, and patch the ring 0 data bases. If a user attempts to call hphcs_ in the normal fashion, hphcs_ is entered into the KST, an SDW is assigned, and access rights are determined from the access control list stored in hphcs_'s parent directory. Since most users would not be on the access control list of hphcs_, access would be denied. Ring 0 gates, however, also have a second segment number assigned from the segment loading table (SLT). This duplication posed no problem on the 645, since SLT SDW's were valid only in the ring 0 descriptor segment. However on the 6180, the KST SDW for hphcs_ would be null access ring brackets 0,0,5, but the SLT SDW would read execute (re) access, ring brackets 0,0,5. Therefore, the penetrator need only transfer to the appropriate absolute segment number rather than using dynamic linking to gain access to any hphcs_ capability. This vulnerability was considerably easier to use than any of the others and was carried though identification, confirmation, and exploitation in less than 5 man-hours total (See Section 3.5).

#### 3.3.4.3 Additional Vulnerabilities

The above mentioned 6180 vulnerabilities have been identified and repaired by Honeywell. The capabilities of the SLT-KST Dual SDW vulnerability were demonstrated to Honeywell on 14 September 1973 in the form of an illegal message to the operator's console at the 6180 site in the Honeywell plant in Phoenix, Arizona. Honeywell did not identify the cause of the vulnerability until March 1974 and installed a fix in Multics System 23.6. As of the time of this publication, additional vulnerabilities have been identified but at this time have not been developed into a demonstration.

## 3.4 Procedural Vulnerabilities

This section describes the exploitation by a remote user of several classes of procedural vulnerabilities. No attempt was made to penetrate physical security, as there were many admitted vulnerabilities in this area. In particular, the machine room was not secure and communications lines were not encrypted. Rather, this section looks at the areas of auditing, system configuration control,[23] passwords, and *privileged* users.

### 3.4.1 Dump and Patch Utilities

To provide support to the system maintenance personnel, the Multics system includes commands to dump or patch any word in the entire virtual memory. These utilities are used to make online repairs while the system continues to run. Clearly these commands are very dangerous, since they can bypass all security controls to access otherwise protected information, and if misused, can cause the system to crash by garbling critical data critical data bases. To protect the system, these commands are implemented by special privileged gates into ring zero. The access control lists on these gates restrict their use to system maintenance personnel by name as authenticated by the login procedure. Thus an ordinary user nominally cannot access these utilities. To further protect the system, the patch utility records on the system operator's console every patch that is made. Thus, if an unexpected or unauthorized patch is made, the system operator can take immediate action by shutting the system down if necessary.

Clearly dump and patch utilities would be of great use to a system penetrator, since they can be used to facilitate his job. Procedural controls on the system dump and patch routines prevent the penetrator from using them by the ACL restrictions and the audit trail. However by using the software vulnerabilities described in section 3.3, these procedural controls may be bypassed and the penetration agent can implement his own dump and patch utilities as described below.

Dump and patch utilities were implemented on Multics using the Unlocked Stack Base and Insufficient Argument Validation vulnerabilities. These two vulnerabilities demonstrated two basically different strategies for accessing protected segments. These two strategies developed from the fact that the Unlocked Stack Base vulnerability operates in ring 4 master mode while the Insufficient Argument Validation vulnerability operates in ring 0 slave mode. In addition, there was a requirement that a minimal amount of time be spent with the processor in an anomalous state - ring 4 master mode or ring 0 illegal code. When the processor is in an anomalous state, unexpected interrupts or events could cause the penetrator to be exposed in a system crash.

#### 3.4.1.1 Use of Insufficient Argument Validation

As was mentioned above, the HIS 645 implementation of Multics simulates protection rings by providing one descriptor segment for each ring. Patch and dump utilities

[23] System configuration control is a term derived from Air Force procurement procedures and refers to the control and management of the hardware and software being used in a system with particular attention to the software update tasks. It is not to be confused with the Multics dynamic reconfiguration capability, which permits the system to add and delete processors and memories while the system is running.

can be implemented using the Insufficient Argument Validation vulnerability by realizing that the ring zero descriptor segment will have entries for segments which are not accessible from ring 4. Conceptually, one could copy an SDW for some segment from the ring 0 descriptor segment to the ring 4 descriptor segment and be guaranteed at least as much access as available in ring 0. Since the segment number of a segment is the same in all rings, this approach is very easy to implement.

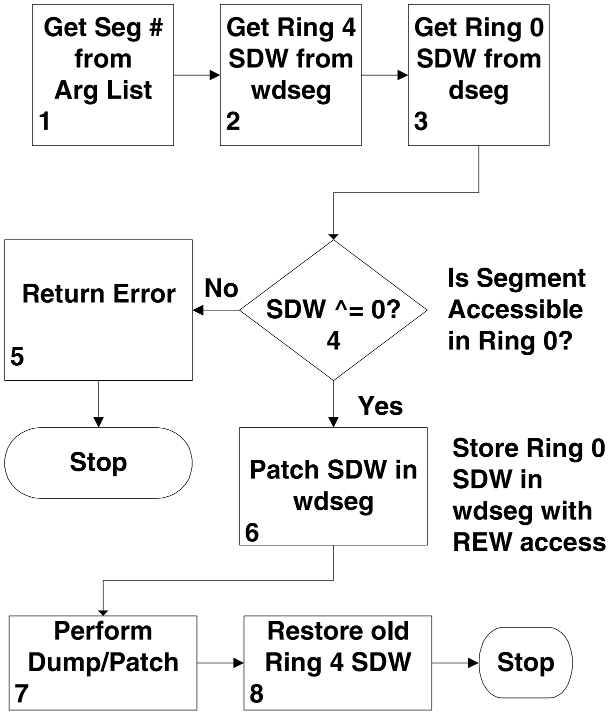


**Figure 11. Dump/Patch Utility Using Insufficient Argument Validation**

The exact algorithm is shown in flow chart form in Figure 11. In block 2 of the flow chart, the ring 4 SDW is read from the ring 4 descriptor segment (wdseg) using the Insufficient Argument Validation vulnerability. Next the ring 0 SDW is read from the ring 0 descriptor segment (dseg). The ring 0 SDW must now be checked for validity, since the segment may not be accessible even in ring 0.[24] An invalid SDW is represented by all 36 bits being zero. One danger present here is that if the segment in question is deactivated,[25] the SDW being checked may be invalidated while it is being manipulated. This event could conceivably have disastrous results, but as we shall see in Section 3.4.2, the patch routine need only be used on segments which are never deactivated. The dump routine can do no harm if it accidentally uses an invalid SDW, as it always only *reads* using the SDW, conceivably reading garbage but nothing else. Further, deactivation of the segment is highly unlikely since the segment is in *use* by the dump/patch routine.

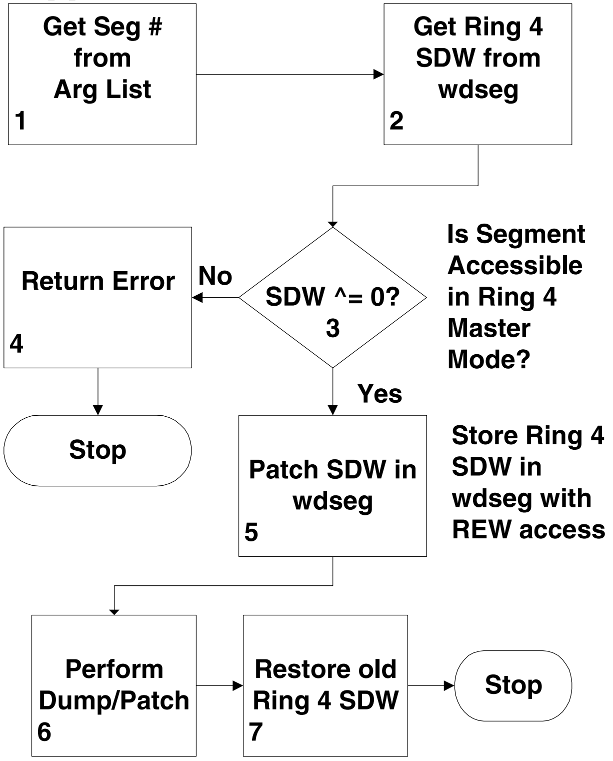


**Figure 12. Dump/Patch Utility Using Unlocked Stack Base**

If the ring 0 SDW is invalid, an error code is returned in block 5 of the flow chart and the routine terminates. Otherwise, the ring 0 SDW is stored into the ring 4 descriptor segment (wdseg) with read-execute-write access by turning on the SDW bits for slave access, write permission, slave procedure (See Figure 2). Now the dump or patch can be performed without using the vulnerability to load or store each 36 bit word being moved. Finally in block 8, the ring 4 SDW is restored to its original value, so that a later unrelated system crash could not reveal the modified SDW in a dump. It should be noted that while blocks 2, 3, 6, and 8 all use the vulnerability, the bulk of the time is spent in block 7 actually performing the dump or patch in perfectly normal ring 4 slave mode code.

#### 3.4.1.2 Use of Unlocked Stack Base

The Unlocked Stack Base vulnerability operates in a very different environment from the Insufficient Argument Validation vulnerability. Rather than running in ring 0, the Unlocked Stack Base vulnerability runs in ring 4 in master mode. In the ring 0 descriptor segment, the segment dseg is the ring 0 segment, and wdseg is the ring 4 descriptor

[24] As an additional precaution, ring 0 slave mode programs run under the same access rules as all other programs. A valid SDW entry is made for a segment in any ring only if the user is on the ACL for the segment. We shall see in Section 3.4.2 how to get around this "security feature".

[25] A segment is deactivated when its page table is removed from core. Segment deactivation is performed on a least recently used basis, since not all page tables may be kept in a core at one time.

segment.[26] However, in the ring 4 descriptor segment, the segment dseg is the ring 4 descriptor segment and wdseg has a zeroed SDW. Therefore, a slightly different strategy must be used to implement dump and patch utilities as shown in the flow chart in Figure 12.[27] The primary difference here is in blocks 3 and 5 of Figure 12 in which the ring 4 SDW for the segment is used rather than the ring 0 SDW. Thus the number of segments, which can be dumped or patched is reduced from those accessible in ring 0 to those accessible in ring 4 master mode. We shall see in Section 3.4.2 that this reduction is not crucial, since ring 4 master mode has sufficient access to provide *interesting* segments to dump or patch.

#### 3.4.1.3 Forging Generation of New SDW's

Two strategies for implementation of dump and patch utilities were shown above. In addition, a third strategy exists which was rejected due to its inherent dangers. In this third strategy, the penetrator selects an unused segment number and constructs an SDW occupying that segment number in the ring 4 descriptor segment using any of the vulnerabilities. This totally new SDW could then be used to access some part of the Multics hierarchy. However, two major problems are associated with this strategy, which caused its rejection. First the absolute core address of the page table of the segment must be stored in the SDW address field. There is no easy way for a penetrator to obtain the absolute address of the page table for a segment not already in his descriptor segment short of duplicating the entire segment fault mechanism which runs to many hundreds or thousands of lines of code. Second, if the processor took a segment or page fault on this new SDW, the ring 0 software would malfunction, because the segment would not be recorded in the Known Segment Table (KST). This malfunction could easily lead to a system crash and the disclosure of the penetrator's activities. Therefore, the strategy of generating new SDW's was rejected.

### 3.4.2 Forging the Non-Forgeable User Identification

In section 2.2.3, the need for the protected, non-forgeable identification of each user was identified. This non-forgeable ID must be compared with access control list entries to determine whether a user may access some segment. This identification is established when the user logs into Multics and is authenticated by the user password.[28] If this user identification can be forged in any way, then the entire login audit mechanism can be rendered worthless.

The user identification in Multics is stored in a per-process segment called the process data segment (PDS). The PDS resides in ring 0 and contains many constants used in ring 0 and the ring 0 procedure stack. The user identification is stored in the PDS as a character string representing the user's name and a character string representing the user's project. The PDS must be accessible to any ring 0 procedure within the user's process and must be accessible to ring 4 master mode procedures (such as the signaller). Therefore, as shown in the Sections 3.4.1.1 and 3.4.1.2, the dump and patch utilities can dump and patch portions of the PDS, thus *forging the non-forgeable user identification*. Appendix E shows the actual user command needed to forge the user identification.

This capability provides the penetrator with an *ultimate weapon*. The agent can now undetectably masquerade as any user of the system including the system administrator or security officer, immediately assuming that user's access privileges. The agent has bypassed and rendered ineffective the entire login authentication mechanism with all its attendant auditing machinery. The user whom the agent is impersonating can login and operate without interference. Even the *who table* that lists all the users currently logged into the system records the agent with his correct identification rather than the forgery. Thus to access *any* segment in the system, the agent need only determine who has access and change his user identification as easily as a legitimate user can change his working directory.

It was not obvious at the time of the analysis that changing the user identification would work. Several potential problems were foreseen that could lead to system crashes or could reveal the penetrator's presence. However, none of these proved to be a serious barrier to masquerading.

First, a user process occasionally sends a message to the operator's console from ring 0 to report some type of unusual fault such as a disk parity error. These messages are prefaced by the user's name and project taken from the PDS. It was feared that a random parity error could "blow the cover" of the penetrator by printing his modified identification on the operator's console.[29] However, the PDS in fact contains two copies of the user identification – one formatted for printing and one formatted for comparison with access control list entries. Ring 0 software keeps these strictly separated, so the penetrator need only change the access control identification.

Second, when the penetrator changes his user identification, he may lose access to his own programs, data and directories. The solution here is to assure that the access control lists of the needed segments and directories grant appropriate access to the user as whom the penetrator is masquerading.

[26] Actually wdseg is the descriptor segment for whichever ring (1-7) was active at the time of the entry to ring 0. No conflict occurs since wdseg is always the descriptor segment for the ring on behalf of which ring 0 is operating

[27] This strategy is also used with the Execute Instruction Access Check Bypass vulnerability, which runs in ring 4.

[28] Clearly more sophisticated authentication schemes than a single user chosen password could be used on Multics (see Richardson [24]). However, such schemes are outside the scope of this paper.

[29] This danger exists only if the operator or system security officer is carefully correlating parity error messages with the names of currently logged in users.

Finally, one finds that although the penetrator can set the access control lists of his ring 4 segments appropriately, he cannot in any easy way modify the access control lists of certain per process supervisor segments including the process data segment (PDS), the process initialization table (PIT), the known segment table (KST), and the stack and combined linkage segments for ring 1, 2, and 3. The stack and combined linkage segments for ring 1, 2, and 3 can be avoided by not calling any ring 1, 2, or 3 programs while masquerading. However the PDS, PIT, and KST are all ring 0 data bases that must be accessible at all times with read and write permission. This requirement could pose the penetrator a very serious problem; but, because of the very fact that these segments must *always* be accessible in ring 0, the system has already solved this problem. While the PIT, PDS and KST are paged segments,[30] they are all used during segment fault handling. In order to avoid recursive segment faults, the PIT, PDS, and KST are never deactivated.[31] Deactivation, as mentioned above, is the process by which a segment's page table is removed from core and a segment fault is placed in its SDW. The access control bits are set in an SDW *only* at segment fault time.[32] Since the system never deactivates the PIT, PDS, and KST, under normal conditions, the SDW's are not modified for the life of the process. Since the process of changing user identification does not change the ring 0 SDW's of the PIT, PDS, and KST either, the penetrator retains access to these critical segments without any special action whatsoever.

### 3.4.3 Accessing the Password File

One of the classic penetrations of an operating system has been unauthorized access to the password file. This type of attack on a system has become so embedded in the folklore of compute security that it even appears in the definition of a security *breach* in DOD 5200.28-M [7]. In fact, however, accessing the password file internal to the system proves to be of minimal value to a penetrator as shown below. For completeness, the Multics password file was accessed as part of this analysis.

#### 3.4.3.1 Minimal Value of the Password File

It is asserted that accessing the system password file is of minimal value to a penetrator for several reasons. First, the password file is generally the most highly protected file in a computer system. If the penetrator has succeeded in breaking down the internal controls to access the password file, he can almost undoubtedly access *every other file in the system*. Why bother with the password file?

Second, the password file is often kept enciphered. A great deal of effort may be required to invert such a cipher, if indeed the cipher is invertible at all.

Finally, the login path to a system is generally the most carefully audited to attempt to catch unauthorized password use. The penetrator greatly risks detection if he uses an unauthorized password. It should be noted that an unauthorized password obtained outside the system may be very useful to a penetrator, if he does not already have access to the system. However, that is an issue of physical security, which is outside the scope of this paper.

#### 3.4.3.2 The Multics Password File

The Multics password file is stored in a segment called the person name table (PNT). The PNT contains an entry for each user on the system including that user's password and various pieces of auditing information. Passwords are chosen by the user and may be changed at any time.[33] Passwords are scrambled by an allegedly non-invertible enciphering routine for protection in case PNT appears in a system dump. Only enciphered passwords are stored in the system. The password check at login time is accomplished by the equivalent of the following PL/I code:

```
if scramble_(typed_password) = pnt.user.password
        then call ok_to_login;
        else call reject_login;
```

For the rest of this section, it will be assumed that the enciphering routine is non-invertible. In a separate volume [15], Downey demonstrates the invertibility of the Multics password scrambler used at the time of the vulnerability analysis.[34]

The PNT is a ring 4 segment with the following access control list:

```
        rw      *.SysAdmin.*
        null    *.*.*
```

Thus by modifying one's user identification to the SysAdmin project as in Section 3.4.2, one can immediately gain unrestricted access to the PNT. Since the passwords are enciphered, they cannot be read out of the PNT directly. However, the penetrator can extract a copy of the PNT for cryptanalysis. The penetrator can also change a user's password to the enciphered version of a known password. Of course, this action would lead to almost immediate discovery, since the user would no longer be able to login.

[30] In fact the first page of the PDS is wired down so that it may be used by page control. The rest of the PDS, however, is not wired.

[31] In Multics jargon, their entry hold switches are set.

[32] In fact, a segment fault is also set in a SDW when the access control list of the corresponding segment is changed. This is done to ensure that access changes are reflected immediately, and is effected by setting faults in all descriptor segments that have active SDW's for the segment. This additional case is not a problem, because the access control lists of the PIT, PDS, and KST are never changed.

[33] There is a major problem that user chosen passwords are often easy to guess. That problem, however, will not be addressed here. Multics provides a random password generator, but its use is not mandatory.

[34] ESD/MCI has provided a 'better" password scrambler that is now used in Multics, since enciphering the password file is useful in case it should appear in a system dump.

### 3.4.4 Modifying Audit Trails

Audit trails are frequently put into computer systems for the purpose of detecting breaches of security. For example, a record of last login time printed when a user logged in could detect the unauthorized use of a user's password and identification. However, we have seen that a penetrator using vulnerabilities in the operating system code can access information and bypass many such audits. Sometimes it is not convenient for the penetrator to bypass an audit. If the audit trail is kept online, it may be much easier to allow the audit to take place and then go back and modify the evidence of wrong doing. One simple example of modification of audit trails was selected for this vulnerability demonstration.

Every segment in Multics carries with it audit information on the date time last used (DTU) and date time last modified (DTM). These dates are maintained by an audit mechanism at a very low level in the system, and it is almost impossible for a penetrator to bypass this mechanism.[35] An obvious approach would be to attempt to patch the DTU and DTM that are stored in the parent directory of the segment in question. However, directories are implemented as rather complex hash tables and are therefore very difficult to patch.

Once again, however, a solution exists within the system. A routine called set_dates is provided among the various subroutine calls into ring 0 which is used when a segment is retrieved from a backup tape to set the segment's DTU and DTM to the values at the time the segment was backed up. The routine is supposed to be callable only from a highly privileged gate into ring 0 that is restricted to system maintenance personnel. However, since a penetrator can change his user identification, this restriction proves to be no barrier. To access a segment without updating DTU or DTM:

1. Change user ID to access segment.
2. Remember old DTU and DTM.
3. Use or modify the segment.
4. Change user ID to system maintenance.
5. Reset DTU and DTM to old values.
6. Change user ID back to original.

In fact due to yet another system bug, the procedure is even easier. The module set_dates is callable, not only from the highly privileged gate into ring 0, but also from the normal user gate into ring 0.[36] Therefore, step 4 in the above algorithm can be omitted if desired. A listing of the utility that changes DTU and DTM may be found in Appendix F.

It should be noted that one complication exists in step 5 – resetting DTU and DTM. The system does not update the dates in the directory entry immediately, but primarily at segment deactivation time.[37] Therefore, step 5 must be delayed until the segment has been deactivated – a delay of up to several minutes. Otherwise the penetrator could reset the dates, only to have them updated again a moment later.

### 3.4.5 Trap Door Insertion

Up to this point, we have seen how a penetrator can exploit existing weaknesses in the security controls of an operating system to gain unauthorized access to protected information. However, when the penetrator exploits existing weaknesses, he runs the constant risk that the system maintenance personnel will find and correct the weakness he happens to be using. The penetrator would then have to begin again looking for weaknesses. To avoid such a problem and to perpetuate access into the system, the penetrator can install *trap doors* in the system which permit him access, but are virtually undetectable.

#### 3.4.5.1 Classes of Trap Doors

Trap doors come in many forms and can be inserted in many places throughout the operational life of a system from the time of design up to the time the system is replaced. Trap doors may be inserted at the facility at which the system is produced. Clearly if one of the system programmers is an agent, he can insert a trap door in the code he writes. However, if the production site is a (perhaps online) facility to which the penetrator can gain access, the penetrator can exploit existing vulnerabilities to insert trap doors into system software while the programmer is still working on it or while it is in quality assurance.

As a practical example, it should be noted that the software for WWMCCS is currently developed using uncleared personnel on a relatively open time sharing system at Honeywell's plant in Phoenix, Arizona. The software is monitored and distributed from an open time sharing system at the Joint Technical Support Agency (JTSA) at Reston, Virginia. Both of these sites are potentially vulnerable to penetration and trap door insertion.

Trap doors can be inserted during the distribution phase. If updates are sent via insecure communications – either US Mail or insecure telecommunication, the penetrator can intercept the update and subtly modify it. The penetrator could also generate his own updates and distribute them using forged stationery.

Finally, trap doors can be inserted during the installation and operation of the system at the user's site. Here again, the penetrator uses existing vulnerabilities to gain access to stored copies of the system and make subtle modifications.

Clearly when a trap door is inserted, it must be well hidden to avoid detection by system maintenance personnel. Trap doors can best be hidden in changes to the binary code of a compiled routine. Such a change is completely invisible on system listings and can be detected only by comparing bit by bit the object code and the compiler listing. However, object code trap doors are vulnerable to recompilations of the module in question.

[35] Section 3.4.5 shows motivation to bypass DTU and DTM.

[36] The user gate into ring 0 contains set_dates, so that users may perform reloads from private backup tapes.

[37] Dates may be updated at other times as well.

Therefore the system maintenance personnel could regularly recompile all modules of the system to eliminate object code trap doors. However, this precaution could play directly into the hands of the penetrator who has also made changes in the source code of the system. Source code changes are more visible than object code changes, since they appear in system listings. However, subtle changes can be made in relatively complex algorithms that will escape all but the closest scrutiny. Of course, the penetrator must be sure to change *all* extant copies of a module to avoid discovery by a simple comparison program.

Two classes of trap doors which are themselves source or object trap doors are particularly insidious and merit discussion here. These are the teletype key string trigger trap door and the compiler trap door.

It has often been hypothesized that a carefully written closed subsystem such as a query system or limited data management system without programming capabilities may be made invulnerable to security penetration. The teletype key string trigger is just one example of a trap door that provides the penetrator with a vulnerability in even the most limited subsystem. To create such a trap door, the agent modifies the supervisor teletype modules at the development site such that if the user types normally, no anomaly occurs, but if the user types a special key string, a dump/patch utility is triggered into operation to allow the penetrator unlimited access. The key string would of course have to be some very unlikely combination to avoid accidental discovery. The teletype key string trap door is somewhat more complex than the trap door described below in Section 3.4.5.2. However, it is quite straightforward to develop and insert with relatively nominal effort.

It was noted above that while object code trap doors are invisible, they are vulnerable to recompilations. The compiler (or assembler) trap door is inserted to permit object code trap doors to survive even a complete recompilation of the entire system In Multics, most of the ring 0 supervisor is written in PL/I. A penetrator could insert a trap door in the PL/I compiler to note when it is compiling a ring 0 module. Then the compiler would insert an object code trap door in the ring 0 module without listing the code in the listing. Since the PL/I compiler is itself written in PL/I, the trap door can maintain itself, *even when the compiler is recompiled.*[38] Compiler trap doors are significantly more complex than the other trap doors described here, because they require a detailed knowledge of the compiler design. However, they are quite practical to implement at a cost of perhaps five times the level shown in Section 3.5. It should be noted that even costs several hundred times larger than those shown here would be considered nominal to a foreign agent.

There is also a variant on the compiler trap door called the initialization trap door. Here, the system initialization code is modified by the penetrator to insert other trap doors as the system is brought up. Such trap doors can be relatively invulnerable to detection and recompilation, because system initialization is usually a very complex and poorly understood procedure.

### 3.4.5.2 Example of Trap Door in Multics

```
check$device_name: proc (a, b, code);
dcl   1 code parameter,
        2 err_code fixed bin (35),
        2 key bit (72) aligned,
        2 ptr1 pointer aligned,
        2 ptr2 pointer aligned;
dcl overlay fixed bin (35) based;

/*   Start of regular code  */
      . . .;
/* Here check$device_name would
   normally return  */
if key = bit_string_constant_password
   then ptr2->overlay = ptr1->overlay;

return;
end check$device_name;
```

**Figure 13. Trapdoor in check$device_name**

To demonstrate the feasibility of inserting and distributing trap doors, the following object code change was made to check$device_name to provide a generalized dump or patch facility. Figure 13 shows a PL/I program that performs essentially the same function as the trap door actually inserted in the object code. The listing of the actual trap door and its calling program is in Appendix C. Check$device_name is normally called with three arguments, the third being a fixed binary error code. The trap door is inserted at the point at which check$device_name would have returned. The trap door looks at the 72 bits immediately following the error code that was passed by the caller. If those 72 bits match a predefined 72 bit password, then the fixed binary word to which ptr1 points is copied into the fixed binary word to which ptr2 points. Since check$device_name is a ring 0 procedure, this copy is carried out using the ring 0 descriptor segment and allows the caller to read or write any word in ring 0. Dump and patch utilities can use this trap door exactly like the Insufficient Argument Validation vulnerability. The 72 bit key is used to ensure that the vulnerability is not invoked by accident by some unsuspecting user.

The actual insertion of the trap door was done by the following steps:

1. Change user identification to project SysLib.
2. Make patch in object archive copy of check$device_name in >ldd>hard>object.
3. Reset DTM on object archive.

[38] This type of trap door does not require a higher level language. Entirely analogous trap doors could be placed in an assembler.

4. Mark patch in bound archive copy of check$device_name in >ldd>hard>bound_components.
5. Reset DTM on bound archive.
6. Reset user identification.

This procedure ensured that the object patch was in all library copies of the segment. The DTM was reset as in Section 3.4.4, because the dates on library segments are checked regularly for unauthorized modification. These operations did not immediately install the trap door. Actual installation occurred at the time of the next system tape generation.

A trap door of this type was first placed in the Multics system at MIT in the procedure del_dir_tree. However, it was noted that del_dir_tree was going to be modified and recompiled in the installation of Multics system 18.0. Therefore, the trap door described above was inserted in check$device_name just before the installation of 18.0 to avoid the recompilation problem. Honeywell was briefed in the spring of 1973 on the results of this vulnerability analysis. At that time, Honeywell recompiled check$device_name, so that the trap door would not be distributed to other sites.

#### 3.4.6 Preview of 6180 Procedural Vulnerabilities

To actually demonstrate the feasibility of trap door distribution, a change which could have included a trap door was inserted in the Multics software that was transferred from the 645 to the 6180 at MIT and from there to all 6180 installations in the field.

### 3.5 Manpower and Computer Costs

Table 3 outlines the approximate costs in man-hours and computer charges for each vulnerability analysis task. The skill level required to perform the penetrations was that of a recent computer science graduate of any major university with a moderate knowledge of the Multics design documented in the **Multics Programmers' Manual** [4] and Organick [22], plus nine months experience as a Multics programmer. In addition, the penetrator was aided by access to the system listings (which are in the public domain) and access to an operational Multics system on which to debut penetrations. In this example, the RADC system was used to test penetrations prior to their use at MIT, since a system crash at MIT would reveal the intentions of the penetrations.[39]

Costs are broken down into identification, confirmation, and exploitations. Identification is that part of the effort needed to identify a particular vulnerability. It generally involves examination of system listings, although it sometimes involves computer work. Confirmation is that effort needed to confirm the existence of a vulnerability by using it in some manner, however crude, to access information without authorization. Exploitation is that effort needed to develop and debug command procedures to make use of the vulnerabilities convenient. Wherever possible, these command procedures follow standard Multics command conventions.

All figures in the table are conservative estimates as actual accounting information was not kept during the vulnerability analysis. However, costs did not exceed the figures given and in all probability were somewhat lower.

The costs of implementing the subverter and inverting the password scrambler are not included, because those tasks were not directly related to penetrating the system. (See Downey[15].) The Master Mode Transfer vulnerability has no exploitation cost shown, because that vulnerability was not carried beyond confirmation.

| | Identification | | Confirmation | | Exploitation | | Total | |
|---|---|---|---|---|---|---|---|---|
| Task | Manhrs | CPU $ | Manhrs | CPU $ | Manhrs | CPU $ | Manhrs | CPU $ |
| Execute Instruction Access Check Bypass | 60 | $150 | 5 | $30 | 8 | $100 | 73 | $280 |
| Insufficient Argument Validation | 1 | $0 | 5 | $30 | 24 | $300 | 30 | $330 |
| Master Mode Transfer | 0.5 | $0 | 2 | $20 | -- | --- | 2.5 | $20 |
| Unlocked Stack Base | 0.5 | $0 | 8 | $50 | 80 | $500 | 88.5 | $550 |
| Forging User ID | 5 | $0 | 5 | $30 | 5 | $90 | 15 | $120 |
| check$device_name Trap door | 5 | $0 | 8 | $50 | 5 | $30 | 18 | $80 |
| Access Password File (Does not include deciphering.) | 1 | $0 | 5 | $30 | 24 | $150 | 30 | $180 |
| Total | 73 | $150 | 38 | $240 | 146 | $1170 | 257 | $1560 |

**Table 3. Cost Estimates**

## 4 CONCLUSION

The initial implementation of Multics is an instance of an uncertified system. For any uncertified system:

a. The system cannot be depended upon to protect against deliberate attack
b. System *fixes* or restrictions (e.g., query only systems) cannot provide any significant improvement in protection. Trap door insertion and distribution has been demonstrated with minimal effort and fewer tools (no phone taps) than any industrious foreign agent would have.

[39] It should be noted that while the MIT system was crashed twice due to typographical errors during the penetration, the RADC system was never crashed.

However, Multics is significantly better than other conventional systems due to the structuring of the supervisor and the use of segmentation and ring hardware. Thus, unlike other systems, Multics can form a base for the development of a truly secure system.

## 4.1 Multics is not Now Secure

The primary conclusion one can reach from this vulnerability analysis is that Multics is not currently a secure system. A relatively low level of effort gave examples of vulnerabilities in hardware security, software security, and procedural security. While all the reported vulnerabilities were found in the HIS 645 system and happen to be fixed by the nature of the changes in the HIS 6180 hardware, other vulnerabilities exist in the HIS 6180.[40] No attempt was made to find more than one vulnerability in each area of security. Without a doubt, vulnerabilities exist in the HIS 645 Multics that have not been identified. Some major areas not even examined are I/O, process management, and administrative interfaces. Further, an initial cursory examination of the HIS 6180 Multics easily turned up vulnerabilities.

We have seen the impact of implementation errors or omissions in the hardware vulnerability. In the software vulnerabilities, we have seen the major security impact of apparently unimportant ad hoc designs. We have seen that the development site and distribution paths are particularly attractive for penetration. Finally, we have seen that the procedural controls over such areas as passwords and auditing are no more than "security blankets" as long as the fundamental hardware and software controls do not work.

## 4.2 Multics as a Base for a Secure System

While we have seen that Multics is not now a secure system, it is in some sense significantly "more secure" than other commercial systems and forms a base from which a secure system can be developed. (See Lipner [21].) The requirements of security formed part of the basic guiding principles during the design and implementation of Multics. Unlike systems such as OS/360 or GCOS in which security functions are scattered throughout the entire supervisor, Multics is well structured to support the identification of the security and non-security related functions. Further Multics possesses the segmentation and ring hardware which have been identified [29] as crucial to the implementation of a reference monitor.

### 4.2.1 A System for a Benign Environment

We have concluded that AFDSC cannot run an open multi-level secure system on Multics at this time. As we have seen above, a malicious user can penetrate the system at will with relatively minimal effort. However, Multics does provide AFDSC with a basis for a *benign* multi-level system in which all users are determined to be trustworthy to some degree. For example, with certain enhancements, Multics could serve AFDSC in a two-level security mode with both Secret and Top Secret cleared users simultaneously accessing the system. Such a system, of course, would depend on the administrative determination that since *all* users are cleared at least to Secret, there would be no malicious users attempting to penetrate the security controls.

A number of enhancements are required to bring Multics up to a two-level capability. First and most important, all segments, directories, and processes in the system should be labeled with classification levels and categories. This labeling permits the classification check to be combined with the ACL check and to be represented in the descriptor segment. Second, an earnest review of the Multics operating system is needed to identify vulnerabilities. Such a review is meaningful in Multics, because of its well structured operating system design. A similar review would be a literally endless task in a system such as OS/360 or GCOS. A review of Multics should include an identification of security sensitive modules, an examination of all gates and arguments into ring 0, and a check of all intersegment arguments in ring 0. Two additional enhancements would be useful but not essential. These are some sort of *high water mark* system as in ADEPT-50 (see Weissman [31]) and some sort of protection from user written applications programs that may contain *Trojan Horses*.

### 4.2.2 Long Term Open Secure System

In the long term, it is felt that Multics can be developed into an open secure multi-level system by restructuring the operating system to include a security kernel. Such restructuring is essential since malicious users cannot be ruled out in an open system. The procedures for designing and implementing such a kernel are detailed elsewhere. [10, 12, 13, 20, 23, 26, 27, 30] To briefly summarize, the access controls of the kernel must always be invoked (segmentation hardware); must be tamperproof (ring hardware); and must be small enough and simple enough to be certified correct (a small ring 0). Certifiability is the critical requirement in the development of a multi-level secure system. ESD/MCI is currently proceeding with a development plan to develop such a certifiably secure version of Multics[1].

## APPENDIX A Subverter Listing[41]

This appendix contains listings of the three program modules which make up the hardware subverter described

[40] In all fairness, the HIS 6180 does proved significant improvements by the addition of ring hardware. However, ring hardware by itself does not make the system secure. Only certification as a well-defined closed process can do that.

[41] The actual listing files from this and all subsequent appendices have been omitted, but can be found in [19].

in Section 3.2.1. The three procedure segments which follow are called subverter, coded in PL/I; access_violations_, coded in PL/I; and subv, coded in assembler. Subverter is the driving routine which sets up timers, manages free storage, and calls individual tests. Access_violations_ contains several entry points to implement specific tests. Subv contains entry points to implement those tests which must be done in assembler.

The internal procedure check_zero within subverter is used to watch word zero of the procedure segment for unexpected modification. This procedure was used in part to detect the Execute Instruction Access Check Bypass vulnerability.

The errors flagged in the listing of subv are all warnings or obsolete 645 instructions, because the attached listing was produced on the 6180.

## APPENDIX B Unlocked Stack Base Listing

This appendix contains listings of the four modules which make up the code needed to exploit the Unlocked Stack Base Vulnerability described in Section 3.3.3. The first two procedures, di and dia, implement step one of the vulnerability – inserting code into emergency_shut-down.link (referred to in the listings as esd.link.) The last two procedures, fi and fia, implement step two of the vulnerability – actually using the inserted code to read or write any 36 bit quantity in the system. Figure 9 in the main text corresponds to di and dia. Figure 10 corresponds to fi and fia. As in Appendix A, obsolete 645 instructions are flagged by the assembler.

## APPENDIX C Trap door in check$device_name Listing

This appendix contains listings of the trap door inserted in check$device_name in Section 3.4.5.2 and the two modules needed to call the trap door. Check$device_name is actually one entry point in the procedure check$device_index. The patches are shown in the assembly language listing of the code produced by the PL/I compilation of check$device_index. Most of the patches were placed in the entry sequence to check$device_index, taking advantage of the fact that PL/I entry sequences contain the ASCII representation of the entry name for debugging purposes. Since the debugger cannot run in ring 0, this is essentially free patching space. Additional patches were placed at each return point from check$device_name, so that the trap door would be executed whenever check$device_name returned to its caller.

Zg is a PL/I procedure which calls the trap door to either read or write any 36-bit word accessible in ring 0. Zg uses zdata, an assembly language routine, to define a structure in the linkage section which contains machine instructions with which to communicate with the trap door.

The trap door algorithm is as follows:

1. Set the bp register to point to the argument rcode. Rcode has been bound to zdata$code in the procedure call from zg and must lie on an odd word boundary.
2. Compare the double word at bp|1 with the key string in the trap door to see if this is a legitimate user calling. If the keys do not match, then just return. If the keys do match, then we know who this is and must proceed.
3. Do an execute double (XED) on the two instruction at bp|3. This allows the caller to provide any instructions desired.
4. The two instructions provided by zdata at bp|3 and bp|5 are ldq bp|5 and stq bp|7. Bp|5 and bp|7 contain pointers to the locations from which to read and to which to write, respectively. These pointers are set in zg.

Finally, the trap door simply returns upon completion of the XED pair.

## APPENDIX D Dump Utility Listing

This appendix is a listing of a dump utility program designed to use the trap door shown in Section 3.4.5 and Appendix C. The program, zd, is a modified version of the installed Multics command, ring_zero_dump, documented in the **MPM Systems Programmers' Supplement** [6]. Zd will dump any segment whose SDW in ring zero is not equal to zero. In addition, zd will not dump the ring zero descriptor segment, because the algorithm used would result in the ring 4 descriptor segment being completely replaced by the ring 0 descriptor segment which could potentially crash the system. Zd will also not dump master procedures, since modifying their SDW's could also crash the system.

## APPENDIX E Patch Utility Listing

This appendix is a listing of a patch utility corresponding to the dump utility in Appendix D. The utility, zp, is based on the installed Multics command patch_ring_zero, documented in the **MPM System Programmers Supplement** [6]. Zp uses the same algorithm as zd in Appendix D and operates under the same restrictions. A sample of its use is shown below. Lines typed by the user underlined.

```
zp pds 660 123171163101 144155151156
660 104162165151 to 123171163101
661 144040040040 to 144155151156
Type "yes" if patches are correct: yes
```

As seen above, the command requests the user to confirm the patch before actually performing the patch. The patch shown above changes the user's project identification from Druid to SysAdmin.

## APPENDIX F Set Dates Utility Listing

This appendix is a listing of the set dates utility described in Section 3.4.4. The get entry point takes a pathname as an argument and remembers the dates on the segment at that time. The set entry point takes no arguments and sets the dates on the segment to the values at the time of the call to the get entry point. Set remembers the pathname as well as the dates and may be called repeatedly to handle the deactivation problem discussed in Section 3.4.4.

## GLOSSARY

**Access**
"The ability and the means to approach, communicate with (input to or receive output from), or otherwise make use of any material or component in an ADP System." [7]

**Access Control List (ACL)**
"An access control list (ACL) describes the access attributes associated with a particular segment. The ACL is a list of user identifications and respective access attributes. It is kept in the directory that catalogs the segment." [3]

**Active Segment Table (AST)**
The AST contains an entry for every active segment in the system. A segment is *active* if its page table is in core. The AST is managed with a least recently used algorithm.

**Argument Validation**
On call to inner-ring (more privileged) procedures, argument validation is performed to ensure that the caller indeed had access to the arguments that have been passed to ensure that the called, more privileged procedure does not unwittingly access the arguments improperly.

**Arrest**
"The discovery of user activity not necessary to the normal processing of data which might lead to a violation of system security and force termination of the activity." [7]

**Breach**
"The successful and repeatable defeat of security controls with or without an arrest, which if carried to consummation, could result in a penetration of the system. Examples of breaches are:

a. Operation of user code in master mode;
b. Unauthorized acquisition of I.D. password or file access passwords; and
c. Accession to a file without using prescribed operating system mechanisms." [7]

**Call Limiter**
The call limiter is a hardware feature of the HIS 6180 which restricts calls to a gate segment to a specified block of instructions (normally a transfer vector) at the base of the segment.

**Date Time Last Modified (DTM)**
The date time last modified of each segment is stored in its parent directory.

**Date Time Last Used (DTU)**
The date time last used of each segment is stored in its parent directory.

**Deactivation**
Deactivation is the process of removing a segment's page table from core.

**Descriptor Base Register (DBR)**
The descriptor base register points to the page table of the descriptor segment of the process currently executing on the CPU.

**Descriptor Segment (DSEG)**
The descriptor segment is a table of segment descriptor words which identifies to the CPU to which segments, the process currently has access.

**Directory**
"A directory is a segment that contains information about other segments such as access attributes, number of records, names, and bit count." [3]

**emergency_shutdown**
"This mastermode module provides a system reentry point which can be used after a system crash to attempt to bring the system to a graceful stopping point." [6]

**Fault Intercept Module (fim)**
The fim is a ring 0 module which is called to handle most faults. It copies the saved machine state into an easily accessible location and calls the appropriate fault handler (usually the signaller).

**Gate Segment**
A gate segment contains one or more entry points used on inward calls. A gate entry point is the only entry in a inner ring that may be called from an outer ring. Argument validation must be performed for all calls into gate segments.

**General Comprehensive Operating Supervisor (GCOS)**
GCOS is the operating system for the Honeywell 600/6000 line of computers. It is very similar to other conventional operating systems and has no outstanding security features.

**HIS 645**

The Honeywell 645 is the computer originally designed to run Multics. It is a modification of the HIS 635 adding paging and segmentation hardware.

**HIS 6180**

The Honeywell 6180 is a follow-on design to the HIS 645. The HIS 6180 uses the advanced circuit technology of the HIS 6080 and adds paging and segmentation hardware. The primary difference between the HIS 6180 and the HIS 645 (aside from performance improvements) is the addition of protection ring hardware.

**hcs_**

The gate segment hcs_ provides entry into ring 0 for most user programs for such functions as creating and deleting segments, modifying ACL's, etc.

**hphcs_**

The gate segment hphcs_ provides entry into ring 0 for such functions as shutting the system down, hardware reconfiguration, etc. Its access is restricted to system administration personnel.

**ITS Pointer**

An ITS (Indirect To Segment) Pointer is a 72-bit pointer containing a segment number, word number, bit offset, and indirect modifier. A Multics PL/I aligned pointer variable is stored as an ITS pointer.

**Known Segment Table (KST)**

The KST is a per-process table which associates segment numbers with segment names. Details of its organization and use may be found in Organick [22].

**Linkage Segment**

"The linkage segment contains certain vital symbolic data, descriptive information, pointers, and instructions that are needed for the linking of procedures in each process." [22]

**Master Mode**

When the HIS 645 processor is in master mode (as opposed to slave mode), any processor instruction may be executed and access control checking is inhibited.

**Multics**

Multics, the Multiplexed Information and Computing Service, is the operating system for the HIS 645 and HIS 6180 computers.

**Multi-Level Security Mode**

"A mode of operation under an operating system (supervisor or executive program) which provides a capability permitting various levels and categories or compartments of material to be concurrently stored and processed in an ADP system. In a remotely accessed resource-sharing system, the material can be selectively accessed and manipulated from variously controlled terminals by personnel having different security clearances and access approvals. This mode of operation can accommodate the concurrent processing and storage of (a) two or more levels of classified data, or (b) one or more levels of classified data with unclassified data depending upon the constraints placed on the systems by the Designated Approving Authority." [7]

**OS/360**

OS/360 is the operating system for the IBM 360 line of computers. It is very similar to other conventional operating systems and has no outstanding security features.

**Page**

Segments may be broken up into 1024 word blocks called pages which may be stored in non-contiguous locations of memory.

**Penetration**

"The successful and repeatable extraction and identification of recognizable information from a protected data file or data set without any attendant arrests." [7]

**Process**

"A process is a locus of control within an instruction sequence. That is, a process is that abstract entity which moves though the instructions of a procedure as the procedure is executed by a processor." [14]

**Process Data Segment (PDS)**

The PDS is a per-process segment which contains various information about the process including the user identification and the ring 0 stack. The PDS is accessible only in ring 0 or in master mode.

**Process Initialization Table (PIT)**

The PIT is a per-process segment which contains additional information about the process. The PIT is readable in ring 4 and writable only in ring 0.

**Protection Rings**

Protection rings form an extension to the traditional master/slave mode relationship in which there are eight hierarchical levels of protection numbered 0 – 7. A given ring N may access rings N through 7 but may only call specific gate segments in rings 0 to N-1.

**Reference Monitor**

The reference monitor is that hardware/software combination which must monitor all reference by any program to any data anywhere in the system to ensure the security rules are followed.

a. The monitor must be tamper proof.
b. The monitor must be invoked for every reference to data anywhere in the system.

c. The monitor must be small enough to be proven correct.

### Segment

A segment is the logical atomic unit of information in Multics. Segments have names and unique protection attributes and may contain up to 256K words. Segments are directly implemented by the HIS 645 and HIS 6180 hardware.

### Segment Descriptor Word (SDW)

An SDW is a single entry in a Descriptor Segment. The SDW contains the absolute address of the page table of a segment (if one exists) or an indication that the page table does not exist. The SDW also contains the access control information for the Segment.

### Segment Loading Table (SLT)

The SLT contains a list of segments to be used at the time the system is brought up. All segments in the SLT come from the system tape.

### Signaller

"signaller is the hardcore ring privileged procedure responsible for signalling all fault and interrupt–produced errors." [6]

### Slave Mode

When the HIS 645 processor is in slave mode, certain processor instructions are inhibited and access control checking is enforced. The processor may enter master mode from slave mode only by signalling a fault of some kind.

### Stack Base Register

The stack base register contains the segment number of the stack currently in use. In the original design of Multics, the stack base was locked so that interrupt handlers were guaranteed that it always pointed to a writable segment. This restriction was later removed allowing the user to change the stack base arbitrarily.

### Subverter

The subverter is a procedure designed to test the reliability of security hardware by periodically attempting illegal accesses.

### Trap door

Trap doors are unnoticed pieces of code which may be inserted into a system by a penetrator. The trap door would remain dormant within the software until triggered by the agent. Trap doors inserted into the code implementing the reference monitor could bypass any and all security restrictions on the systems. Trap doors can potentially be inserted at any time during software development and use.

### WWMCCS

WWMCCS, the World Wide Military Command and Control System, is designed to provide unified command and control functions for the Joint Chiefs of Staff. As part of the WWMCCS contract for procurement of a large number of HIS 6000 computers, a set of software modifications were made to GCOS, primarily in the area of security. The WWMCCS GCOS security system was found to be no more effective that the unmodified GCOS security, due to the inherent weaknesses of GCOS itself.

## REFERENCES

1. *ESD 1973 Computer Security Developments Summary*, MCI-74-1, December 1973, HQ Electronic Systems Division: Hanscom AFB, MA.

2. *IBM System/360 Operating System Service Aids*, GC28-6719-0, June 1970, IBM Corporation.

3. *Multics Users' Guide*, AL40, Rev. 0, November 1973, Honeywell Information Systems, Inc.: Waltham, MA.

4. *The Multiplexed Information and Computing Service: Programmers' Manual*, Revision 14, 30 September 1973, Massachusetts Institute of Technology and Honeywell Information Systems, Inc.: Cambridge, MA.

5. *Summary of the H6180 Processor*, 22 May 1973, Information Processing Center, Massachusetts Institute of Technology: Cambridge, MA.

6. *System Programmers' Supplement to the Multiplexed Information and Computing Service: Programmers' Manual*, 1973, Massachusetts Institute of Technology and Honeywell Information Systems, Inc.: Cambridge, MA.

7. *Techniques and Procedures for Implementing, Deactivating, Testing, and Evaluating Secure Resource-Sharing ADP Systems*, DoD 5200.28-M, January 1973, Department of Defense: Washington, DC.

8. *WWMCCS Security System Test Plan*, 23 May 1972, Joint Technical Support Activity, Defense Communications Agency: Washington, DC.

9. Anderson, J.P., *AF/ACS Computer Security Controls Study*, ESD-TR-71-395, November 1971, James P. Anderson and Co., Fort Washington, PA, HQ Electronic Systems Division: Hanscom AFB, MA.

10. Anderson, J.P., *Computer Security Technology Planning Study*, ESD-TR-73-51, Vols. I and II, October 1972, James P. Anderson and Co., Fort Washington, PA, HQ Electronic Systems Division: Hanscom AFB, MA. URL: http://csrc.nist.gov/publications/history/ande72.pdf

11. Andrews, J., M.L. Goudy, *et al.*, *Model 645 Processor Reference Manual*, revision 4, 1 April 1971, Cambridge Information Systems Laboratory, Honeywell Information Systems, Inc.: Cambridge, MA.

12. Bell, D.E. and L.J. LaPadula, *Secure Computer Systems: A Mathematical Model*, ESD-TR-73-278, Vol. II, MTR-2547, Vol. II, November 1973, The MITRE Corporation, Bedford, MA, HQ Electronic Systems Division: Hanscom AFB, MA.

13. Bell, D.E. and L.J. LaPadula, *Secure Computer Systems: Mathematical Foundations*, ESD-TR-73-278, Vol. I, MTR-2547, Vol. I, November 1973, The MITRE Corporation, Bedford, MA, HQ Electronic Systems Division: Hanscom AFB, MA.

14. Dennis, J.B. and E.C. Van Horn, *Programming Semantics for Multiprogrammed Computations.* Communications of the ACM, March 1966. 9(3): p. 143-155.

15. Downey, P.J., *Multics Security Evaluation: Password and File Encryption Techniques*, ESD-TR-74-193, Vol. III, June 1977, HQ Electronic Systems Division: Hanscom AFB, MA.

16. Goheen, S.M. and R.S. Fiske, *OS/360 Computer Security Penetration Exercise*, WP-4467, 16 October 1972, The MITRE Corporation: Bedford, MA.

17. Graham, R.M., *Protection in an Information Processing Utility.* Comm. ACM, May 1968. 11(5): p. 365-369.

18. Inglis, W.M., *Security Problems in the WWMCCS GCOS System*, Joint Technical Support Activity Operating System Technical Bulletin 730S-12, 2 August 1973, Defense Communications Agency: Washington, DC.

19. 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

20. Lipner, S.B., *Computer Security Research and Development Requirements*, MTP-142, February 1973, The MITRE Corporation: Bedford, MA.

21. Lipner, S.B., *Multics Security Evaluation: Results and Recommendations*, ESD-TR-74-193, Vol. I, MTR-3267, October 1978, The MITRE Corporation, Bedford, MA, HQ Electronic Systems Division: Hanscom AFB, MA.

22. Organick, E.I., *The Multics System: An Examination of Its Structure*. 1972, Cambridge, MA: The MIT Press.

23. Price, W.R., *Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems*, PhD thesis 1973, Carnegie-Mellon University: Pittsburgh, PA.

24. Richardson, M.H. and J.V. Potter, *Design of a Magnetic Card Modifiable Credential System Demonstration*, MCI-73-3, December 1973, HQ Electronic Systems Division: Hanscom AFB, MA.

25. Saltzer, J.H., *Protection and the Control of Information Sharing in Multics.* Comm. ACM, July 1974. 17(7): p. 388-402.

26. Schell, R.R., P.J. Downey*, et al.*, *Preliminary Notes on the Design of Secure Military Computer Systems*, January 1973, HQ Electronic Systems Division: Hanscom AFB, MA. URL: http://csrc.nist.gov/publications/history/sche73.pdf

27. Schiller, W.L., *Design of a Security Kernel for the PDP-11/45*, ESD-TR-73-294, MTR-2709, December 1973, The MITRE Corporation, Bedford, MA, HQ Electronic Systems Division: Hanscom AFB, MA.

28. Schroeder, M.D. and J.H. Saltzer, *A Hardware Architecture for Implementing Protection Rings.* Comm. ACM, March 1972. 15(3): p. 157-170.

29. Smith, L.A., *Architectures for Secure Computing Systems*, ESD-TR-75-51, MTR-2772, April 1975, The MITRE Corporation: Bedford, MA, HQ Electronic Systems Division, Hanscom AFB, MA.

30. Walter, K.G., W.F. Ogden*, et al.*, *Primitive Models for Computer Security*, ESD-TR-74-117, 23 January 1974, Case Western Reserve University, Cleveland, OH: HQ Electronic Systems Division, Hanscom AFB, MA.

31. Weissman, C. *Security Controls in the ADEPT-50 time sharing system.* in Fall Joint Computer Conference. 1969, Vol. 35. AFIPS Conference Proceedings, AFIPS Press, Montvale, NJ. p. 119-133.