--

Practical Problems with a Cryptographic Protection Scheme

Jonathan M. Smith

Distributed Systems Laboratory
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

ABSTRACT

Z *is a software system designed to provide media-transparent network services on a collection of* U*NIX*® *machines. These services are comprised of file transfer and command execution;* **Z** *preserves file ownership on remote transfer, and more significantly, owner and group identity when executing commands remotely. In order to secure known vulnerabilities in the system, enhancements were made. In particular, a cryptographically-derived checksum was added to the messages. After the initial implementation of the checksumming scheme, several iterations of performance improvement occurred. The result was unsatisfactory to the user community, so the checksum was removed. Instead, vulnerabilities were reduced by improved monitoring and maintenance procedures.*

1. Introduction

1.1. History

Z was initially implemented *circa* 1978 in order to cope with an ever-increasing number of UNIX systems at a large industrial computation center. The environment was becoming unmanageable; unmanageable in the sense that it was difficult to administer the systems in a controlled and consistent manner. It was clear that some mechanism which allowed a user to operate in a true multi-system environment was necessary. However, there was no consistent network organization. There were a variety of subnetworks of various reliabilities and bandwidths, these included bus-to-bus, channel-to-channel, and synchronous remote job entry (RJE) links. RJE served as a fully-connected network (albeit

---

This work was supported, in part, by Bell Communications Research, (Morristown, NJ) under Project DAWN.

a slow one) as all systems were connected to some mainframe system for various services, e.g., bulk printing. The Network Systems Corporation HYPERChannel™ bus is a very high speed (*circa* 50Mbits/sec) device that allows machines to be interconnected in a local area network. In late 1981, an NSC HYPERchannel began to connect all the systems, and due to its bandwidth, became the primary media for **Z** communication. **Z** provided an easy to use and uniform interface to the network; the physical media used in the transport is transparent to the user. Use of the media was optimized by a statically-calculated bandwidth-weighted best-path selection scheme coupled with dynamically-calculated reliability data.

1.2. Architecture

Z is viewed by a large segment of its user population as a UNIX command with which they accomplish various tasks across several computing systems. In actuality, **Z** is a large collection of both loosely and tightly coupled cooperating software modules, distributed across all machines on the network. The command invocation is the first link in a long chain of events.

The **Z** command line syntax specifies file transfer or remote execution. The standard input of the local **Z** execution can be used to provide input for the remote command. The **Z** semantics preserve user ids on the remote system, both for file permissions and command execution. One or more systems can be specified as destinations, and aliasing is available for compact naming of subsets of the available systems.

The **Z** system provides facilities for examining queued jobs, retrying jobs when failures occur, removing corrupted files, notifying senders upon error or job completion, and secondary routing on link failure. Various internal machine-to-machine interfaces were devised to mask incompatibilities between heterogeneous processor families.

The basic architecture of the system is illustrated in figure 1.

1.2.1. Local Actions

The **Z** command serves as a gateway to the lower level transport layers of the system. It "packetizes" the data to be transported, storing such information as can only be gathered at invocation time, such as the current directory and identity of the invoking user. If a file name was given as an argument, **Z** ensures that the file is accessible, and saves other information, such as the file's ownership, the access permissions, and the file size in characters. Also on the command line can be a list of one or more destination systems; some validity checking is done on these names. **Z** can also take a command string as an argument.

Based on the arguments and the identity information, **Z** builds a header, which is used by other modules. Any data to be transmitted is then appended to the header. The bound data "packet" is passed to a "gatekeeping" module, **Z**qer. **Z**qer is invoked as the last action of the **Z** command with the *exec()* system call. It takes the formed data packet and enqueues it in a known spool directory, where the transport control module, **Z**demon,

will find it when woken by a signal from **Z**qer. **Z**qer contains knowledge of a spooling directory, spooling and sequencing protocols, and interprocess communication procedures. Since these actions are required both by local invocations of **Z** and by the remote receiving node, good software engineering suggested a common module.

Zdemon is a process which constantly waits for an event to occur: this event is the signal that **Z**qer sends to it, alerting it to the fact that there is work to do. When there are no files to process in its spool directory, it remains in an idle state, waiting on an "event", the "signal" sent to it by the **Z**qer process. When **Z**demon receives this signal, it scans the known spool directory, looking for the work which should have just appeared. When started by a command line, or when woken by an *alarm* signal or a signal from **Z**qer, **Z**demon searches the well-known directory where invocations of **Z**qer have placed the packetized data.

Zdemon examines the header information of the enqueued packet to determine its next action. If the packet is to be processed locally, a **Z**xmit0 is spawned. The **Z**xmit0 carries out the local action; it is given the file as a parameter, and determines how to process it from the enqueued header.

If the packet is to be passed on to another node, the **Z**demon selects a transmission medium. A transmission route to the destination system is selected, and based on this route, some lower level transport mechanism is used by *fork()* -ing a **Z**xmit sub-process of the appropriate flavor. The **Z**xmit job manages the details of transporting the packet to the remote system and ensures its correct receipt. It uses the underlying transport mechanism both to actually transmit the data and to perform certain actions at the destination. Among these actions is the execution of a copy of the program **Z**recv.

1.2.2. Remote Actions

All of the transport mechanisms which the **Z** system uses provide at least a minimal form of remote command execution. When a **Z** job arrives at a destination UNIX machine, it generates a **Z**recv process. **Z**recv executes **Z**qer to let **Z**demon know that there is work to be done. Since this is now a "local" job, **Z**demon executes **Z**xmit0 to carry out the specified action. Effectively, **Z**recv serves as a "friend" to the traveling process: it appears as if **Z**recv is a local-to-local invocation of **Z**, requested on behalf of the remote (sending) machine. This design is modular and elegant; job-handling is correct with respect to local or remote destinations, while being ignorant, for the most part, of the details.

The details of transmission are handled by **Z**xmit modules, which prepare a message for transmission over their respective media, and proceed to carry out the transmission. All of these modules must provide some mechanism for invoking the **Z**recv module on the remote system once the data/ command packet has arrived.

These modules are perhaps the lowest "layer" of the **Z** architecture, since unlike the others, they have to concern themselves with details of the communications link, such as file size limitations. In most cases, the **Z**mit modules merely invoke commands which are provided as part of a link's operational subsystem. If the RJE medium is used, intermediary, non-UNIX system "hops" may have to be made.

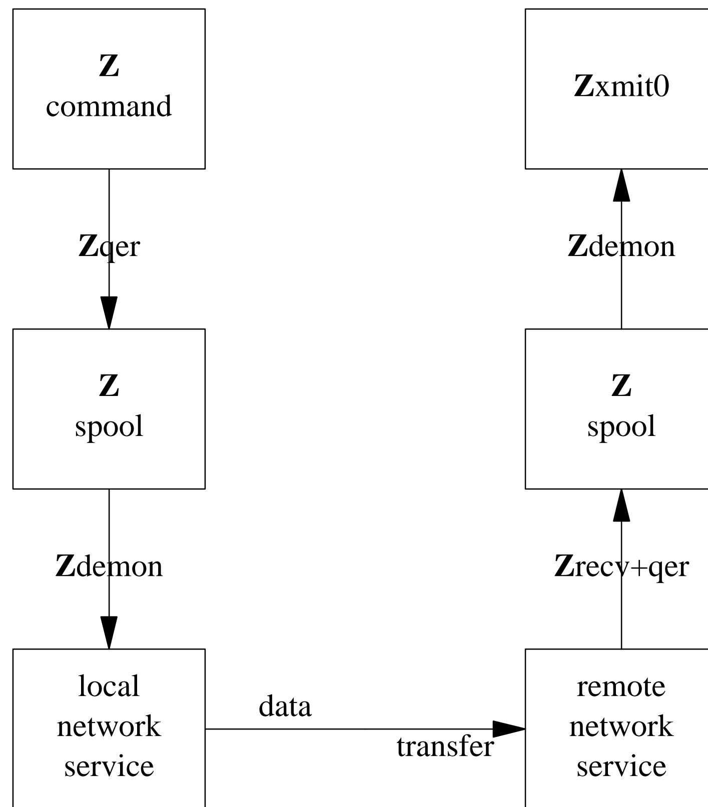

Figure 1: Z Architecture

So that **Z** can maintain user id's across systems, several modules must possess super-user (unlimited file access) privileges. While passive interception is potentially dangerous, as it provides *information,* we were rather more concerned with active interlopers; those who intend to *modify* data and/or commands.

1.3. Security Problems

In 1983, we became concerned about the security of **Z**, and immediately recognized several potential vulnerabilities. These stemmed from several architectural features, as can be gleaned from figure 1. The fact that the system preserves user identity is the major reason for a security threat. First, the ability to execute commands remotely means that a breakin on one system can be extended to others. Second, the complete interconnectivity provided by **Z** meant that a breakin on one system could be extended to all of the machines. If a **Z** packet could be modified enroute to its destination, then the user id or any message contents could be set to interesting values. Thus, the points of vulnerability [8] were those which had the potential for message alteration.

First, the spool had to be kept secure, or otherwise files with arbitrary contents could be written. Second, the local network services had to be kept secure. This was more of a problem than it appeared; these systems often spooled jobs internally, and their access-control strategies were not easy to change. To verify the claims we made about the lack

of security inherent in the system, we obtained root permissions from an ordinary account. This was done through altering a file spooled by the RJE mechanism. The file resided in the UNIX spool for about 0.5 second, and was enciphered with a simple modification of a Caesar scheme. Unfortunately, the software preserved user ownership of the spooled file, so that a user could modify the file. This was necessary due to the design of the RJE software. Breaking the cipher was trivial, and by repeatedly sending messages and polling, a file was captured, modified, and transmitted to its (unsuspecting!) destination. Simply protecting this spool directory and modifying the RJE software was insufficient; the RJE jobs were spooled on the mainframe as well, where we could not guarantee security.

2. A Server-based solution

After studying the problem, we came to some conclusions:

1. While administrative control was not completely ours (as was the case with intermediate mainframe systems) the system was at risk.
2. We were far less interested in protecting against traffic analysis than in protecting against modification.
3. Sending enciphered messages was undesirable for several reasons, including (1) recovery from errors, (2) use of intermediate nodes needing source and destination data, and (3) system status reporting.
4. In addition, requirements were that the user interface could not change, e.g., by requesting a password.

After examining the literature on data security [5, 2] we decided that the right approach was to use a cryptographic *checksum* in order to detect data modification. The checksum is computed by encrypting the data and then computing a checksum from the encrypted text. Thus, the messages could be sent in cleartext, with a checksum prepended to the header. Modifications could be detected, and modified messages discarded. The improved error-detection was a byproduct. Since changes to either the header (uids) or the message (binaries for system programs) were dangerous, the entire packet had to be involved in the checksum. The initial implementation used a 32-bit checksum.

A variety of encipherment schemes were examined, and experimental implementations were done to evaluate the performance of the schemes. Even after implementation in assembly language, a cryptosystem using large primes consumed unacceptable amounts of CPU time, even for very short strings. DES [4, 5] was examined, but once again the throughput of the implementation was insufficient. While it is clear that DES is intended to be implemented in hardware, the chips available at the time were expensive and slow. In addition, we had three architectures to contend with, and kernel changes would have been necessary. We sped up the UNIX library implementation of DES by a factor of 3 using hand-optimized code and small assembly-language routines. Recent research [1] indicates that speedups up to a factor of 20 or more can be accomplished by applying some mathematical sophistication in the software implementation. Our speedup reduced the CPU time required for encrypting a one megabyte file from about 2340

seconds on an AT&T 3B20S™ (the 3B20S is roughly comparable to a DEC VAX™ 11/780) to about 830 seconds. Execution of a simple command which counts the characters in a file requires about 5 seconds of CPU time, so the contribution of file reading code is low. Considerable computation was necessary to convert byte-oriented files to bitstreams of one bit per character. Use of techniques such as cipher-block chaining would slow an implementation down further.

Bishop's factor of 20 speedup should reduce this time to about 120 seconds of CPU time. Unfortunately, **Z** was often used for transfer of files which were up to a megabyte in size, and response times (comprised of CPU times and delays caused by scheduling, processor sharing, and I/O) measured in minutes were unacceptable. We finally decided that a modification of the UNIX *crypt* command would be the best solution. Even though *crypt* has recently been shown to be insecure, the rotor ciphers, once set up, allow extremely rapid encipherment to take place. While our work preceded Reeds and Weinberger's [7], we seem to have anticipated some of the elements of their approach; we varied the rotor-shifting steps in a password- dependent way in order to frustrate analysis of blocks of text for which one of the rotors remains fixed.

Since we were convinced that encryption technology would improve, we wanted to add the encipherment to the system in such a way that new solutions, e.g., DES chips, could be incorporated easily.

2.1. Encryption Server

The change in the architecture is illustrated in figure 2.

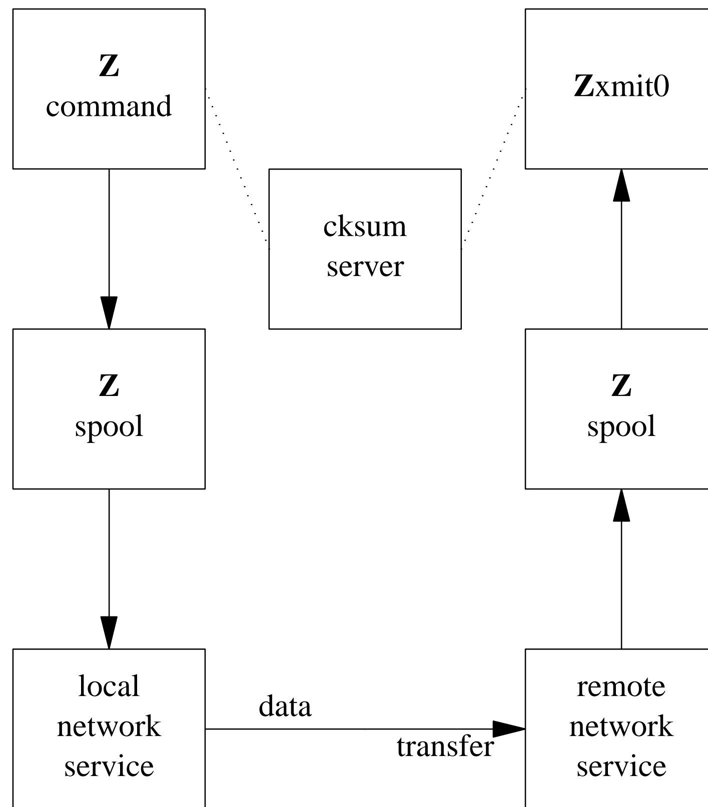

Figure 2: Z Architecture with checksum server

The server was passed a filename argument using a secure FIFO queue. The filename was for the packet which **Z** had just gathered. Insertion in the queue by a process woke the server, which enciphered the file, computed the checksum, and passed back the result to the calling process. The passwords were per-system, stored in a secure file, which the server checked frequently for modification. The passwords were encrypted using DES [6], and served as seeds to the rotor generation for the Enigma-clone. If the file hadn't changed, the encryption was cached, for performance reasons. It's clear that a public-key system would have been more effective for this task, but the available systems performed too poorly.

The idea of the server architecture was to emulate the semantics of a remote procedure call. In this way, the server process could be transparently replaced by another server process with the same functionality which used hardware encryption or other tricks to get better response times.

2.2. Problems

The basic design of the server was well thought out, and in another systems environment, may still be the right way to go, because the advantages in terms of software engineering are manifold, e.g. modularity, information-hiding, et cetera. Unfortunately, encryption is a CPU-intensive activity; hence, the UNIX system scheduler assigns the server process

lower and lower priority as time goes on, and it becomes "slower" with respect to response time. Improved scheduling technology could remedy this problem, but it was neither available nor administratively desirable, except for our application's use.

Since the process (**Z**) waiting for a reply (the checksum) cannot assume that the server is up, it can time out on the write to the server's request queue, using the *alarm()* facility. While it can retry, if the server is sufficiently slow the request may not be serviced ‘‘in time’’. If this occurs, the packetizing software will assume server failure and therefore fail to packetize the request. On the other hand, if timeouts are not facilitated, the software may appear to be so slow that users will find alternate means of data transport. For large files and a busy server, the response times were measured in minutes.

In addition, the server proved to be an administrative nightmare: it was hard to understand without a great deal of expertise in encipherment, systems programming, and network software; it created ‘‘mysterious’’ files when it wasn't keeping up with the request queueing rate; and it was dependent on the sanity of several files. Consequently, a re-design was done which preserved most of the positive features of the server design, while improving response time and reducing administrative effort.

3. Re-design, no server

The major goal of the redesign effort was (initially) to increase the reliability of the server process, and hence reduce the administrative effort. After a painstaking analysis of the alternatives, it was decided that the server module should be removed and the encryption services be provided by in-line code rather than interprocess communication. While on the one hand, the burden of performing a DES encryption on the cleartext password could not be shared between users of a server, the following were true:

- A small data transfer would not be penalized in real-time response for following a large data transfer in the request queue. (The large transfer would cause an external server to accumulate a large amount of CPU time, thus penalizing it in the scheduling discipline.)
- The DES encryption is not necessary, provided that any cleartext password is encrypted before being put into a secured file.
- Per system passwords were eliminated, as this proved to be of little use in practice. One password is used for all systems on a **Z** network; eliminating a system would then require changing the password on all machines except the one to be cut off.
- While re-engineered to be robust, the interprocess communication was complex, and slow in response time. Putting the code in-line permitted several optimizations, which led to significant performance improvements, a factor of 3 to 5.

The necessary code to perform checksumming and encrypting of file contents was moved inline. The *Pack()* and *UnPack()* calls were designed so that callers would be ignorant of their methodology; this proved to be true in practice, as not one line of the calling modules had to be re-coded to reflect the fact that the server had been eliminated. The interprocess communication code was eliminated and the *CheckSum()* call made directly. The encryption algorithm was modified to remove a reflecting rotor from the encipherment

process, thus removing a memory access and two arithmetic operations from the process of encrypting a byte. This was done without reducing the cryptographic strength of the algorithm, as the reflecting rotor mainly aids decipherment. The checksumming routine was also re-coded for greater efficiency; the net result was that **Z** required about 20 seconds of CPU time to queue a 1 megabyte file on an AT&T 3B-20S. A command line (no file access) requires a little more than a second of CPU on the same machine. The additional CPU overhead in each invocation is therefore directly proportional to the resource utilization of the request: this seemed fair. Unfortunately, much **Z** use is administrative, and a traffic analysis showed that the average packet size was about 100K bytes, implying about 2 seconds of response time penalty for using encryption. We felt that this was acceptable, but extensive testing with the user community raised vocal complaints. Thinking these spurious, we surveyed the user community, and concluded that the encryption feature, weakened as it was through:

1. Lack of public key technology,
2. Use of a cipher system known to be breakable, and
3. Increasing dependence on protected files,

was no longer viable.

4. Conclusions

There were several benefits which accrued from our work. The rewriting of the software resulted in a more robust, readable, and elegant system. Various dangling pointer errors were corrected, and buffer size checks were added; this serves to remove other obscure security problems. The desire for end-to-end encryption, or something close, led to a complete, and better redesign for the packet header; it was re-encoded entirely in ASCII. This resulted in better portability in a heterogeneous machine environment. The calls to the encryption routines were commented out of the source code, a total of 4 lines of "C"; no other modifications were necessary. Several utility programs had either been inappropriately placed in the directory hierarchy or gave inappropriate levels of privilege to users. These were changed.

Administrative rigor was applied to reduce the security threat. Vulnerable directories were checked carefully for permissions; they are now monitored on a regular basis. As newer networking technologies are phased in, the old methods, such as RJE, which used potentially unsafe intermediary nodes, are being phased out. Careful administration is used [3] at present to prevent surprises.

Our application points out some of the serious problems with applying cryptographic technology in practice. First, while a cryptographic checksum is the obvious solution, it was clearly an afterthought, and had to be added to an existing architecture. Second, the uses of the system can put severe performance constraints on an otherwise workable system. Our problem was the relatively frequent transfer of large data files, and the demands of timesharing users for good response times. We tried to cure this by weakening the cryptographic checksum scheme, but the results were unacceptable, so security is maintained primarily by administrative vigilance.

-- --

However, when the networking technologies such as Ethernet™ are broadcast media, ‘‘promiscuous-listeners’’, and more seriously, ‘‘modifiers’’, start to resurface as an issue. Thus, the increased computational cost of creating a system with a higher cryptographic ‘‘work factor’’ begins to seem more reasonable. The economics of cost/performance might justify special-purpose hardware, e.g., for DES encipherment. The tradeoffs between security and response-time should be examined carefully, and frequently.

In particular, a useful and productive area of research would be one which resulted in a set of curves which related cryptographic strength to some useful performance metric. One such metric, alluded to in this paper, is the number of arithmetic operations required per byte of a large file. The analysis represented by the performance curve allows a system designer to compare systems and select an appropriate system for the application. Without such analysis, most cryptographic work is likely to remain of interest mainly to mathematicians; practical work requires getting the details right.

5. Notes

® UNIX is a Registered Trademark of AT&T Bell Laboratories.

3B20 is a trademark of AT&T.

VAX is a trademark of Digital Equipment Corporation.

HYPERChannel is a trademark of Network Systems Corporation.

Ethernet is a trademark of Xerox Corporation.

6. References

[1] Matt Bishop, ‘‘An Application of a Fast Data Encryption Standard Implementation,’’ *Computing Systems* **1**(3), pp. 221-254 (1988).

[2] D.R. Denning, *Cryptography and Data Security,* Addison-Wesley (1982).

[3] F. T. Grampp and R. H. Morris, ‘‘UNIX Operating System Security,’’ *AT&T Bell Laboratories Technical Journal* **63**(8, Part 2), pp. 1649-1672 (October 1984).

[4] A. G. Konheim, *Cryptography: A Primer,* Wiley-Interscience, New York (1981).

[5] C. Meyer and S. Matyas, *Cryptography: A New Dimension in Computer Data Security,* Wiley-Interscience (1982).

[6] R. Morris and K. Thompson, ‘‘UNIX Password Security,’’ *Communications of the ACM* **22**, pp. 594-597 (November 1979).

[7] J. A. Reeds and P. J. Weinberger, ‘‘File Security and the UNIX System Crypt command,’’ *AT&T Bell Laboratories Technical Journal* **63**(8, Part 2), pp. 1673-1684 (October 1984).

[8] D. M. Ritchie, ‘‘On the Security of UNIX,’’ in *UNIX Programmer’s Manual, Section 2* (1983). AT&T Bell Laboratories