The following paper was originally published in the
Proceedings of the Fifth USENIX UNIX Security Symposium
Salt Lake City, Utah, June 1995.

Network Randomization Protocol: A Proactive Pseudo-Random Generator

Chee-Seng Chow and Amir Herzberg
IBM T.J. Watson Research Center
Yorktown Heights, New York

Network Randomization Protocol:

A Proactive Pseudo-Random Generator

Chee-Seng Chow Amir Herzberg

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

{cschow,amir}@watson.ibm.com

Abstract

A major security threat to any security solutions based on a centralized server is the possibility of an adversary gaining access to and taking control of the server. The adversary may then learn secrets, corrupt data, or send erroneous messages. In practice, such an adversary may be more prevalent than one would like to admit. It may be a malicious hacker, a virus in an application program, or an unscrupulous system administrator.

Proactive security is a novel approach to the server security problem. It uses the distribution of data and control to multiple servers and periodic refreshes between servers. By distributing data and control, one or more servers may be compromised without compromising the system. Periodic refreshes between servers allow a compromised server to "recover" after the attacker leaves, thereby contributing to the system security. A fraction (in some cases all) of the servers must be compromised *simultaneously* in order to compromise the system.

This paper describes the *Network Randomization Protocol (NRP)* — a proactive protocol for generating cryptographically secure pseudo-random numbers. The protocol is designed for operation in the Internet and includes defenses against clogging attacks. Issues related to the design and implementation of the protocol are discussed.

As virtually no cryptographic task is possible without a source of randomness or pseudo-randomness, NRP is an important basic building block for many cryptographic functions. Furthermore, it serves to illustrate the main ideas and intuitions of proactive security.

Keywords: cryptography, proactive security, network protocol, pseudo-random generator, Internet security, client-server.

1 Introduction

1.1 Server Insecurity

Computers are often the main targets in security attacks against computing systems. The security problem becomes worse in a network environment. The "system" is no longer a mainframe housed in a physically secure room but consists of many geographically distributed machines linked together by a communication network. Important data and vital functions are delegated to one or more servers, which are not always physically secure.

While communication links in a computer network are also subject to security attacks, the attacks are handled by standard cryptographic techniques such as encryption and authentication. However, security threats against computers (servers, in particular) are not readily dealt with. The attacks could be in the form of a virus in an application program, a malicious hacker, or an unscrupulous system operator. The attacks may occur intermittently but over a long period result in significant loss of secret information, corruption of vital data, and disruption of services. One reason why security threats against servers are not easily addressed is that such threats, especially internal threats, are hard to formalize. Few good solutions are known.

1.2 Survey of Existing Solutions

We briefly review some existing solutions to the problem of server security.

One approach is to use secure hardware. The approach assumes that a certain component of the system (e.g., where secret keys are store) is physically secure against intruders and system operators. The security of the system depends on the security of the component. However, such systems tend to be expensive, hard to service, and proprietary. The design and implementation of the secure hardware is often not open to public review since its security may depend on its secrecy. In this paper, we will not discuss hardware-based solutions. Instead we will focus on software-based solutions, which rely on cryptographic techniques.

The Unix password security system [MT79] relies on the difficulty to invert a one-way function, f. Instead of storing a user password, the system stores f(password). To login, the user supplies a password. The system authenticates the user by applying f to the supplied password and checking the result against the password file. This solution ensures that even if an adversary gains access to the password file, she cannot masquerade as the user.

However, the solution assumes that the communication between the user and the system is secure. Otherwise, an adversary can obtain the password by eavesdropping. A modified solution has been proposed by Lamport to handle this problem [Lam81].

More recently, Bellovin and Merritt [BM93] have a solution that ensures a user password remains secure even if an attacker has access to the server database. Furthermore, an eavesdropper cannot mount a dictionary attack to guess the user password. This work is an extension of their earlier work [BM92].

It is clear from this (non-exhaustive) survey of server security solutions that server break-ins are a major security concern. The common approach of all the above solutions is to avoid keeping "secret" in the server. Nevertheless, an active attacker can disrupt services to selected clients by corrupting the server database or masquerade as a server to unsuspecting clients.

1.3 Proactive Security

We now look at a different approach to server security, where there is secret information to be protected.

One solution is to do secret sharing [Sha79] on multiple servers. This approach makes sense especially if the system already has multiple servers for load-sharing. However traditional secret sharing has the following drawback: An adversary can compromise one server at a time until sufficient number of servers are compromised to reveal the secret.

This is how proactive security comes in. The two key ideas of proactive security are: (1) the distribution of data and control to multiple servers, and (2) periodic refreshes between the servers. Each server is initialized with an initial secret (share). By distributing data and control to multiple servers, a fraction of the servers may be compromised without compromising the system. The periodic refreshes allow a compromised server to *recover* (regain secrecy) after an attacker leaves. The refresh period is a security parameter; it could be minutes, hours, days, weeks, or months depending on the application and the security desired. It is a tradeoff between security and performance.

Since a compromised server can recover and contribute to the overall system security, in order to compromise the system, a fraction (in some cases all) of the servers must be compromised *simultaneously*. In particular, an attacker can compromise one server at a time until all servers have been compromised (though not in the same time period) and yet the system remains secure. The notion of *recovery* is a key property of proactive security.

The idea of a *mobile* adversary has been used by others [OY91] in another context. It does not model all attacks, but captures a large class of real-life attacks. Some examples of such "transient" attacks are malicious hackers (who carry out attacks from the network during wee hours), untrustworthy system operators (who snoop during their shifts), and viruses (which are removed when the system periodically reboots).

We call the approach *proactive security* because the refreshes are send periodically independent of whether the system is under attack or not. In particular, they are not sent *in reaction* to attacks or suspected attacks. While a server is compromised, refreshes from its neighbors do not help. But as soon as the adversary leaves, the refreshes help the server recover. By sending the refreshes periodically, the system does not need to know whether a server is compromised or not, since detecting an attack is often harder than preventing one.

In practice, servers often have the same flaws. While it is true that an adversary may exploit the same weaknesses to break into multiple servers, proactive security makes this significantly harder, since the adversary must break into the servers simultaneously. Of course, one can also improve security by making sure that the servers are sufficiently different in architecture, located far apart, and administered by different groups of people.

We and others in our group are currently exploring various applications of proactive security. Some possible applications are: (1) public key certification and signature authority (2) authentication and key distribution center (3) server-server key maintenance (4) pseudo-random number generation.

In this paper, we discuss the application of proactive security to the problem of generating pseudo-random numbers. This is the simplest application; it is also the best understood. More importantly, we have actually implemented the solution and have dealt with the implementation issues. This simple but important application serves well to illustrate the main ideas and the power of proactive security.

1.4 Pseudo-Random Number Generation

We briefly review the problem of pseudo-random number generation and the importance of randomness in security. For a more detailed discussion, see [ECS94] for an excellent treatment of the topic.

Modern security systems increasingly rely on cryptography for security assurances. However, the security of many cryptographic algorithms and protocols depends on a continuous source of random numbers. Some crucial applications of such numbers are in the generation of cryptographic keys, key renewal, nonce generation, and so on. In particular, the Unix password solution [MT79] and its modification [Lam81] both require randomness for password generation. The works of Bellovin and Merritt [BM92, BM93] also require a random source. Virtually no cryptographic task is possible without a source of randomness or pseudo-randomness.

The most direct means of getting random numbers is through special hardwares. As discussed earlier, special hardwares are expensive, non-portable, and usually proprietary. Moreover, most existing computers, including many security servers, are not equipped with such hardwares. There are many pitfalls in using supposedly random sources in a computer (such as disk access times and system clocks) as a source of randomness. (See [ECS94].)

In contrast, software solutions are cheap, portable, and are repeatable (an essential property in testing and debugging). The problem of generating numbers with random properties in software is well-studied. (See, e.g., [Knu81].) Such numbers are *pseudo-random,* in contrast to *random* numbers generated from physical random sources such as shot-noise or quantum devices. A basic requirement of pseudo-random numbers is that they have similar statistical properties to random numbers.

For security applications, we are interested in the generation of *cryptographically secure* pseudo-random numbers. A computationally bounded adversary cannot distinguish such numbers from random numbers (except for some negligible advantage). In particular, the numbers should appear *unpredictable* to the adversary.

To generate cryptographically secure pseudo-random numbers, the program must hold a secret key (seed) unknown to the adversary. (The algorithm is "public knowledge".) If only one server is used and the server is compromised, then the numbers generated are no longer secure since the adversary can also generate them (using the same seed as the server). This suggests a proactive approach to the problem.

1.5 Network Randomization Protocol

Network randomization protocol (NRP) is a practical adaptation of the theoretical protocol in [CH94]. Whereas [CH94] is *synchronous,* assumes *a fully connected topology,* and is *based on pseudo-random functions,* NRP is *asynchronous,* allows *arbitrary topology,* and is *based on pseudo-random generators.* Unlike [CH94] which also provides reconstructibility, the sole purpose of NRP is randomization.

Similar in concepts to the Network Time Protocol, which provides time services using a group of servers for synchronization, NRP provides (cryptographically secure) pseudo-random numbers using a group of servers. Each server is initialized with a randomly (or pseudo-randomly chosen) seed and periodically generates and sends pseudo-random values (refreshes) to its neighbors. Upon receiving a refresh, the server updates its seed. NRP provides a simple, uniform way to integrate different sources of randomness (*local* such as disk-access time, user-keyboard time and *remote* such as network delay or random values from other servers).

NRP is specifically designed for the Internet environment. In this environment, it is easier for an adversary to break into a server or to carry out active attacks against a server than to eavesdrop or to intercept *all* messages to a server. Messages may arrive at the server via different routes. The protocol is also applicable to other network environments as well. Finally, NRP is designed to run as a daemon process, as we do not expect servers dedicated for running NRP.

1.6 Contributions

This paper introduces the key ideas of proactive security from a system perspective. The discussion is informal and intuitive. We show that the ideas are practical, efficient, and simple by presenting an in-depth discussion on the design and implementation of NRP. The discussion is distilled from our experience implementing the protocol on an IBM RS/6000 workstation running AIX.

1.7 Organization

The paper is organized as follows: In section 2 we discuss the adversary model, introduce a modified pseudo-random generator, and describe the randomization protocol. In Section 3, we discuss the design and implementation of the server, server-server communications, and some practical extensions to the protocol. Section 4 describes the client-server interface and addresses some related security issues. Finally, we summarize and discuss some conclusions in Section 5.

2 Basic Concepts

2.1 Adversary Model

Throughout this paper we make the standard computational complexity assumptions in cryptography such as certain problems cannot be efficiently solved (e.g., in polynomial time) and that adversaries are computationally bounded. All actions by the adversaries or servers occur in polynomial time.

When servers are compromised, the adversaries have total control of the servers. Adversaries can learn secret information, corrupt critical data, crash servers, and make them send erroneous messages to other servers. The adversaries can do all these in a fully coordinated manner.

In addition, the adversaries are *mobile*. An adversary can move from one server to another. When an adversary leaves, the server reverts to the original program, though corrupted program data remain corrupted. (This is to model a large class of attacks that are transient.) Without such an assumption, the notion of an adversary leaving a server would not make sense.

Another type of attacks are "clogging attacks", which deny service to the server by overwhelming the server with messages. In general such attacks, are extremely difficult, if not impossible, to prevent. The best that one can do is to log the occurrences for a system administrator to handle offline and to limit the maximal work in response to a message.

Communication links between servers are not immune from attacks. Links may be compromised, in which case an adversary can read, remove, alter, or inject messages. We assume that injecting a message is easier than eavesdropping (which is often the case in Internet).

Servers can protect their communications through cryptographic techniques such as encryption and authentication. However, all these techniques require a server to keep some secret keys. Consequently when a server is compromised all communications into and out of the server are also compromised. Furthermore, a malicious adversary may corrupt the keys, resulting in communication breakdowns even though the physical links are fully operational. Fortunately, as we will see in Section 3, NRP does not need encryption or authentication.

2.2 Pseudo-Random Generator

The network randomization protocol is based on a modified pseudo-random generator to be described in this section. The discussion is informal and intuitive. Proofs and a more formal treatment are deferred to another paper.

A traditional pseudo-random generator [BM84] is a function that when given a secret seed outputs a stream of bits that appear random to the adversary. In particular, the adversary cannot guess an unseen bit better than chance (plus a negligible advantage) after observing other output bits. The adversary also cannot guess the secret seed better than chance (plus a negligible advantage). In practice, there are efficient implementations which are believed to be pseudo-random. For example, one can use the output of the DES-CBC encryption function, with the secret seed as encryption key, on some input string.

For our purpose, we use a modified pseudo-random generator (PRG) that consists of an s-bit internal seed variable and a traditional pseudo-random generator function. To limit the damage when the server is compromised, the seed is updated whenever an output is generated.

The following operations are supported by the PRG: **PRG-Create**, **PRG-Free**, **PRG-Get-value**, and **PRG-Update-seed**. The first function **PRG-Create** takes as input an s-bit pseudo-random value, instantiates a PRG, and initializes the seed with the input. The second function releases system resources used by the PRG. Of interest are the last two functions.

The function **PRG-Get-value** when invoked (after **PRG-Create**) outputs an ℓ-bit pseudo-random value and updates the internal seed. A possible implementation of the function is as follows: The seed is used by the traditional generator function to generate $s+\ell$ pseudo-random bits, where ℓ bits are output. The remaining s bits are used to update the seed.

Let f_k denote the traditional pseudo-random generator with seed k, and (o_1, o_2) denote the output of f_k, where o_1 and o_2 are of lengths s and ℓ, respectively. When **PRG-Get-Value** is invoked, the PRG is updated according to the following equations:

$$\begin{aligned}(o_1, o_2) &\leftarrow f_k \\ k &\leftarrow o_1,\end{aligned}$$

and **PRG-Get-Value** outputs o_2.

The function **PRG-Get-value** should have the following properties:

A.1 An adversary cannot guess an output value better than chance (plus some negligible advantage) by observing other outputs. In particular, an adversary cannot guess any seed values better than chance (plus some negligible probability).

A.2 If the seed is revealed to the adversary at some time, then the adversary cannot guess any prior unseen outputs of **PRG-Get-value** better than chance (plus some negligible advantage).

Property A.2 limits the amount of information revealed when a server is compromised. In particular, the adversary cannot deduce its prior pseudo-random outputs from the seed.

Refreshes from other servers provide new randomizations to a server. The pseudo-random values from other servers are incorporated into the PRG using **PRG-Update-seed**, which takes an s-bit input. **PRG-Update-seed** *does not* simply replace the seed with the input but combine them in such a way that the following properties hold:

A.3 If an adversary does not know the seed, then no matter what sequence of updates $u_1, u_2, \ldots, u_N$ an adversary chooses and applies to the PRG (using **PRG-Update-seed**), the seed of the PRG remains pseudo-random. The adversary may invoke any number of **PRG-Get-value** in between **PRG-Update-seed**.

A.4 If an adversary knows the seed and chooses all of the updates $u_1, u_2, \ldots, u_i, \ldots, u_N$, except u_i which is a pseudo-random value unknown to the adversary, then the PRG regains its pseudo-randomness when updated with u_i. The adversary may invoke any number of **PRG-Get-value** in between **PRG-Update-seed**. (Note that all the updates except u_i may be known to the adversary; they may be functions of u_i.)

Property A.3 ensures that a server PRG remains pseudo-random even if an adversary has complete control of all communications into a server.

Property A.4 (is often stronger than needed but) ensures that a single refresh that evades the adversary allows a previously compromised server to regain pseudo-randomness. This property handles replay attacks where the adversary resends an update without knowing the value (e.g., the value is encrypted). Note that simply taking the exclusive or of the seed with the update would not satisfy Property A.4, since an adversary may resend u_i, without knowing u_i, to nullify it.

A possible implementation of **PRG-Update-seed** is as follows: An s-bit value is formed by taking the exclusive or of the input with the current seed. The value is used in the traditional pseudo-random generator function to generate an s-bit value which will be the new seed. If **PRG-Update-seed** is given the update u, PRG is updated as follows:

$$\begin{aligned}k &\leftarrow k \oplus u \\ (o_1, o_2) &\leftarrow f_k \\ k &\leftarrow o_1.\end{aligned}$$

(Note that this is simply invoking **PRG-Get-value** once, after taking the exclusive or of the update with the current seed.)

2.3 Network Randomization Protocol

Network randomization protocol (NRP) runs on a group of servers. The subset of servers that a server refreshes are its *neighbors.* For simplicity, we will assume that the graph of neighborhood relationship is symmetric and connected.

2.3.1 Update Protocol

Each server has its own PRG. At the beginning, the PRG in every server is initialized with an independently and pseudo-randomly chosen seed. Periodically, a server sends to each of its neighbors a pseudo-random value, the output of **PRG-Get-value**. When a server receives a refresh, it updates its seed using **PRG-Update-seed** with the refresh as input. Each server runs the update protocol independently, possibly with a different refresh period.

The state of a server is completely determined by its seed and the refreshes received. (For simplicity, we have assumed that the sending of refreshes within a period is an atomic operation.) To predict the state of the server, an adversary must know the seed and monitors all refreshes into the server. If the adversary misses a single refresh, the server will regain its pseudo-randomness.

2.3.2 Properties

NRP have following properties:

B.1 If all, but one, servers are compromised, then the uncompromised server remains secure no matter what refreshes are sent to it.

B.2 If an adversary knows the states of all servers and knows all messages in transit except for one refresh, then the unseen refresh can re-randomize the entire network.

Intuitively, it is easy to see that Properties B.1 and B.2 follow from Properties A.3 and A.4 of the PRG, respectively. A more formal treatment and proofs of the properties are deferred to another paper.

3 Server Design

This section describes some practical aspects related to the design and implementation of an NRP server. The security of server-server communications and some practical extensions to the protocol are also discussed. We begin with a review of the design objectives.

3.1 Design Objectives

The objectives are to keep the server simple and efficient. To minimize processing, no detailed accounting of refreshes is kept. Refreshes are sent using the unreliable datagram protocol (UDP) for efficiency. The idea is that pseudo-random values are cheap to produce (cf. Property A.2) and are readily incorporated (cf. Property A.3).

One of the states of a server is its seed. The goal is to avoid introducing new states to the server. This is especially important in proactive security since it limits what an adversary can learn or corrupt when the server is compromised; it also simplifies the recovery process when the adversary leaves.

3.2 Server-Server Communications

At first it seems that cryptographic means are needed to ensure confidentiality and authentication of server-server communications. This is not an appealing prospect as it introduces shared keys between neighbors. Beside being computationally expensive, it introduces states in the server.

Fortunately, the encryption and authentication of refreshes are not needed. They do not improve security in our model. The reasons are as follows: We consider two cases. First, if the adversary has complete information about a server and monitors all communications into the server, it can perfectly predict the server states. No cryptographic processing in the server can help. Second, if the adversary does not know some secret information in the server (e.g., an encryption key), then that information could have been incorporated into the PRG (using **PRG-Update-seed**) and no matter what an adversary does to the communication links, the server remains secure.

Some of the data in a refresh are as follows: a message-type field, a positive count to indicate the number of pseudo-random values in the message, an array of pseudo-random values. More details on the message format will be provided in another document [CH95].

3.3 Sources of Randomness

To improve randomization, NRP can (and should) be used in conjunction with any locally available random sources. Three interfaces are provided for this purpose: (1) message IPC queue (2) UDP port (3) function linkage.

The first two methods involve sending a message to the server. The last method requires implementing and linking with the server a function that gets values from local random sources. At initialization, the operator can also input a "password" to generate the initial seed for the PRG.

For further randomization, we provide a program that "samples" micro-second clock readings. The sampler provides periodic local randomness to the server.

Normally the system clock would be a poor source of randomness. A novelty here is the interaction between the server and the random sampler (which runs as a separate Unix process). The sampler gets random values from the server (using the client-server interface to be described in the next section) and combines them with the clock readings (using DES for mixing).

The sampler sends random values to the server using the message IPC queue. The server updates its pseudo-random generator using refreshes received from the sampler. (Refreshes from the sampler are treated no different from refreshes from other servers.) This interaction and feedback between the sampler and the server can be viewed as a local version of NRP.

3.4 Practical Extensions

There are some simple, practical extensions to improve the security of the protocol.

To make it harder for an adversary to monitor communications, a server can send refreshes probabilistically (e.g., send a refresh only if the parity of a pseudo-random value is even). And to exploit the randomness of network delays, the sending of refreshes can be spread out over the refresh period. This allows incoming refreshes to interleave with outgoing refreshes. By interleaving invocations of **PRG-Get-value** with **PRG-Update-seed**, the server makes it much harder for an adversary to track its states. Since refreshes are sent unreliably, the loss of refreshes over the links is an additional source of randomness.

To detect clogging attacks, the server checks that the number of UDP messages received in a period is below some threshold. When an attack is detected, the server raises an alarm and logs all relevant messages. The final determination and handling of the attack is left to the system administrator.

4 Client-Server Interface

We now discuss the design and implementation of the client-server interface. The interface allows clients to get pseudo-random values from their local server. Security issues related to malicious clients are also addressed.

4.1 Modified Adversary Model

In this section we consider a modified adversary model where the adversary has *partial* control of a server machine. In a multi-user, multi-process environment such as Unix, the adversary may control a client process without controlling the operating system or the server process. Such an adversary may eavesdrop or disrupt client-server communications, masquerade as a client or as a server, and attack the server or other clients within the limitations imposed by the operating system.

4.2 Design Objectives

For clients to regain security after an adversary leaves (i.e., to be proactive), they should periodically get pseudo-random values from the server. A simple function call interface, which hides the interprocess communication (IPC) details, is provided.

The objectives are to protect the server and the clients from malicious clients with partial control of the system and to keep the server stateless. The design of the interface is complicated because Unix does not provide secure IPC mechanisms. Extra efforts are needed to authenticate messages between processes.

4.3 Application Programming Interface

We design a simple, secure application programming interface (API) for client applications to get pseudo-random values from the server. The interface is implemented using a PRG (pseudo-random generator). Instead of directly using pseudo-random values obtained from the server, each client has its own PRG. Pseudo-random values from the server are used to update the PRG. This provides proactive security to clients and at the same time reduces communication overhead.

We would like to design the interface similar to the standard C-library pseudo-random functions **srand** and **rand** (Note that this library is not cryptographically secure.) However our interface is necessarily more complicated because it hides the IPC details.

The API provides the following functions: **NRP-Create**, **NRP-Free**, and **NRP-Get-value**.

NRP-Create creates and initializes a PRG by requesting a pseudo-random value from the server. The client can also provide a pseudo-random value for additional randomization.

NRP-Free performs the reverse operations, cleans up and releases resources to system.

NRP-Get-value gets pseudo-random outputs from the PRG (using **PRG-Get-value**). Periodically it uses a secure IPC interface to get pseudo-random values from the server to randomize the PRG (using **PRG-Update-seed**).

4.4 Secure IPC from Server to Client

We now describe the IPC mechanism between clients and the server.

A general solution to ensure secure client-server communication is to introduce shared keys between server and clients. However, the server doesn't need any keys, since only clients need to obtain secure pseudo-random values from the server. Therefore it suffices for the server to provide a unique pseudo-random value to each client at the beginning of service. A client can then use the value to initialize its PRG.

The IPC message interface between client and server is implemented as follows: The server has a well-known IPC queue that only it can read but can be written by other processes. To keep the server stateless, the server sends a pseudo-random value (from **PRG-Get-value**) in response to a request from a client. To reduce the effect of clogging attacks, the server process is not interrupted when an IPC message arrives on its queue. Instead, the server handles the requests when it wakes up to send periodic refreshes. (An adversary can still clog the server input queue with spurious messages.) The server raises an alarm and logs the appropriate messages when the number of requests exceeds some threshold.

All book-keeping needed to correlate reply to request is handled by the interface. The client creates a private reply queue that only it can read from and other processes can write to. The reply queue ID is included in the request to the server. The client can either wait for a reply from the server (in which case **NRP-Get-value** would block) or retrieve the reply at a subsequent invocation of **NRP-Get-value**. The queue is destroyed when the reply is received. It remains to show how the client authenticates a reply from the server.

4.5 Authentication of Server

A well-known authentication technique is the random challenge-response protocol. In this protocol a client puts a pseudo-random value in the request. The server puts the same value in the reply. The client authenticates the reply by checking the value in the reply. Since only the server and the client can read their respective queues, the protocol should work. However, it doesn't because the client may not be able to generate a secure pseudo-random value in the first place.

The solution we implemented uses the System V message IPC facility and the Unix file protection. First the server creates a file that only it can write to but can be read by others. The server writes its process ID in the file. When a client receives an IPC message, it first checks that there is exactly one message in its queue. Since a queue is used only once for each IPC request, more than one message in the queue indicates an attack. The client obtains the process ID of the sender through the message IPC facility and authenticates the server by checking the ID against the file.

4.6 Shared Memory Interface

The message IPC mechanism is expensive. Therefore this interface is used at the beginning and infrequently thereafter. For better performance and better resiliency against clogging attacks, we provide a more efficient, less secure interface.

The simplest interface is to use shared memory for the server to refresh the clients. When a client invokes **NRP-Create**, a shared memory region is allocated and initialized. The server periodically updates the shared memory region with pseudo-random values. The client reads from the region to update its PRG, whenever **NRP-Get-value** is invoked.

However, in Unix it is not practical to provide a private shared region between the server and each client. (There may be thousands of clients.) Moreover, the design also introduces states in the server.

A better design is to use a single shared memory region that only the the server can write to but can be read by all the clients. The server does not need to keep track of its clients and thus remains stateless. The server periodically writes an array of pseudo-random values in the shared memory region. Clients can then randomly choose some combination of values from the region to update their PRGs. By simultaneously writing multiple pseudo-random values in the shared region, the server provides additional randomization with minimal additional costs.

We note that a malicious client can continuously monitor the entire shared region to learn the updates used by other clients. However, this is not sufficient for the adversary to learn the states of other clients, since the IPC interface ensures that the initial seeds in the PRGs of the other clients are secure.

4.7 Final Design

We now summarize the final design of the client-server interface. Each client uses its own PRG to generate pseudo-random values. The interface uses a combination of IPC messages and shared memory to obtain pseudo-random values from the server to update the PRG.

Periodically or after some number of invocations of **NRP-Get-value**, the interface reads from the shared memory to update the PRG. At initialization (and infrequently thereafter), a secure, but more expensive, message IPC mechanism is used to obtain private pseudo-random values from the server directly. (In this design, the server can run as an ordinary Unix process.)

To detect clogging attacks, the server checks that the number of IPC messages in its queue is below some threshold. When an attack is detected, the server raises an alarm, logs the relevant messages, and leaves the handling of the attacks to the system administrator.

The design has been implemented on AIX. The implementation should be easily ported to other

Unix systems. It should be possible to port to other multi-user, multi-process operating systems with basic file protection and IPC mechanisms.

5 Summary and Conclusions

Proactive security introduces periodic refreshes to the traditional notions of distributed control and secret sharing. Some of the novel properties of a proactive system are the robustness and resiliency against powerful mobile adversaries, which model real-life security threats such as hackers, untrustworthy system administrators, viruses, and rogue programs. In practice, the threats are often less mobile and less coordinated.

We presented an in-depth discussion on the design and implementation of NRP (Network Randomization Protocol). The problem of generating cryptographically secure numbers is important and nontrivial. NRP may not completely solve the problem but makes it significantly harder for an adversary to compromise the system. In practice, most people would not implement stand-alone NRP servers, but would combine the protocol with other security functions served by these servers.

An interesting practical question is whether NRP can produce strong pseudo-random numbers using only weak random sources in the servers (e.g., clock drifts, disk access times, and message delays). A related theoretical question is whether NRP can produce pseudo-randomness using only "weak" PRGs.

NRP is simple, lightweight, robust, and has many interesting properties. More work is needed to formalize and prove these properties. It is a useful basic building blocks for higher-level cryptographic applications. Beside fulfilling an important cryptographic function, NRP serves well to illustrate the principles and the power of proactive security.

Availability

NRP is available by anonymous ftp from `software.watson.ibm.com` in `/pub/security/nrp`. The code was developed and tested on AIX 3.2.5 using IBM xlc compiler and has been ported to SunOS 4.1.2 using GNU gcc compiler.

References

[BM84] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Computing*, pages 850–864, 1984.

[BM92] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: password based protocols secure against dictionary attacks. In *Proc. IEEE Computer Society Symp. on Research in Security and Privacy*, pages 72–84, May 1992.

[BM93] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: a password based protocol secure against dictionary attacks and password file compromise. In *1st ACM Conference on Computer and Communications Security*, pages 244–250, November 1993.

[CH94] Ran Canetti and Amir Herzberg. Maintaining security in the presence of transient faults. *Crypto*, pages 425–438, 1994.

[CH95] Chee-Seng Chow and Amir Herzberg. Network randomization protocol: A proactive pseudo-random generator. *Internet Working Draft — In preparation*, 1995.

[ECS94] Donald E. Eastlake, Stephen D. Crocker, and Jeffrey I. Schiller. Randomness requirements for security. *Internet RFC 1750*, 1994.

[Knu81] Donald Knuth. *The Art of Computer Programming Vol. 2: Seminumerical Algorithm.* Addison-Wesley, 1981.

[Lam81] Leslie Lamport. Password identification with insecure communication. *Communications of the ACM*, pages 770–772, 1981.

[MT79] R.H. Morris and K. Thompson. Unix password security. *Communications of the ACM*, 22:594, 1979.

[OY91] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks. In *Proceedings of the* 10^{th} *Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada*, pages 51–59, 1991.

[Sha79] Adi Shamir. How to share a secret. *ACM*, 22(11):612–613, November 1979.