An Encrypted Payload Protocol and Target-Side Scripting Engine

Dino A. Dai Zovi
Dino.DaiZovi@twosigma.com

Abstract

Modern exploit payloads in commercial and open-source penetration testing frameworks have grown much more advanced than the traditional shellcode they replaced. These payloads permit interactive access without launching a shell, network proxying, and many other rich features. Available payload frameworks have several limitations, however. They make little use of encryption to secure delivery and communications, especially in earlier stage payloads. In addition, their richer features require a constant network connection to the penetration tester, making them unsuitable against mobile clients, such as laptops, PDAs, and smart phones.

This work introduces a first-stage exploit payload that is able to securely establish an encrypted channel using ElGamal key agreement and the RC4 stream cipher. The key agreement implementation requires only modular exponentiation and RC4 also lends itself to an implementation requiring a very small amount of executable code. This secure channel is used to send further executable code and deliver a fully-featured interpreter to execute mission logic written in the high-level Lua scripting language. This scripting environment permits secure code delivery as well as disconnected operation and execution of penetration testing mission logic.

1 Introduction

Although much work has gone into detecting, mitigating, and preventing their exploitation, remote code execution vulnerabilities remain one of the most prevalent and serious vulnerability classes in software today. While reported remote code execution vulnerabilities in common server software have gotten more rare, they are still highly prevalent in web browsers, office suites, media players, anti-virus suites, and other local application software.

Remote code execution vulnerabilities are exploited by injecting a small amount of executable machine code into the remote process and then triggering the vulnerability (*the injection vector*) in order to cause the remote process to execute the injected code (*the payload*). Traditionally, these payloads have been written in processor-specific assembly language, assembled, and extracted by hand into reusable machine code components. These payloads were typically small and executed an operating system shell, causing them to be commonly referred to as *shellcode*. Common shellcode variants included functionality such as restoring dropped privileges, breaking out of `chroot` environments, and attaching the shell to an inbound or outbound network connection. This style of payload construction was both labor and skill intensive.

With the growth of commercial penetration testing services and especially commercial penetration testing products, exploit payloads have gotten considerably more capable and complex. These payloads have facilitated the construction of products that help less experienced penetration testers better demonstrate the risk presented by exploitable security vulnerabilities. In addition, the increased prevalence of layered host security defenses require that payloads be more robust and capable of more complex logic. The Unix `chroot` system call is typically used to run a network service in a restricted file system environment. Extensions including FreeBSD's `jail` [1] extend this model to assign a dedicated IP address to a *jail*. More recent developments include the Windows Integrity Mechanism in Windows Vista. The Windows Integrity Mechanism is a Biba Integrity Model[2] Mandatory Access Control security system used to implement Internet Explorer's "Protected Mode" and User Account Control (UAC). Protected Mode Internet Explorer runs as a *low-integrity* process that prevents it from writing to anywhere except designed places on the file system or in the registry, regardless of the discretionary access control list on the objects. Escalating privileges in these restricted permission environments, pivoting through networked hosts, or achieving mission objectives from these restricted execution environments requires complex payload logic.

This work addresses two shortcomings of the existing exploit payloads found in commercial and open-source penetration testing tools. The first is a lack of secure payload delivery and communication mechanisms. Most available systems offer partial solutions to this at the present time by either only supporting encryption in later-stage executable agents or by only supporting simple XOR-based traffic encoding. The second area that this work addresses is the applicability of existing payloads against mobile clients. Most of the existing frameworks require a constant network connection to the compromised target in order to carry out mission objectives. This paper introduces a target-side scripting environment in order to push mission-objective logic onto the target. The secure channel established by the earlier stage payload ensures that mission objective logic is not compromised in transit to the target.

This paper is organized as follows. First, a summary of existing advanced payload techniques and frameworks is presented in the next section. Afterwards, Encrypted Payload Protocol describes a system for cryptographically securing exploit payload delivery and communication. The final stage of this payload system, a lightweight and powerful remotely scripted agent is introduced in the section titled Target-Side Scripting. Finally, the paper concludes with some discussion of the security and utility of the described payload system.

2 Related Work

As mentioned above, the development of commercial penetration testing applications and layered host security defenses have significantly increased the complexity of exploit payloads. This section describes the exploit payload systems found in the major available commercial and open-source penetration testing frameworks.

Syscall proxying[3] is a simple and flexible technique for simulating remote execution. Syscall proxying implements a fat client, thin server remote procedure call protocol based on transferring stack frames. To execute a remote system call, the syscall proxy client marshalls the required argument register values and memory buffers into a single buffer. The syscall proxy server writes this marshalled stack buffer directly to its stack segment, pops the general register values from it, and executes the system call specified. After the system call returns, the server pushes all the general purpose registers onto the stack and transfers the stack buffer back to the client. This general mechanism requires very little server-side logic, but provides a great deal of flexibility, allowing the remote client to open files, make network connections, and even exploit other vulnerabilities on the same or other remote systems.

There are some limitations, however, with the syscall proxy's simple server-side logic. For example, it cannot handle the `fork` system call. In addition, file or socket input and output with the `read` or `write` system calls requires transferring the buffer twice because allocating stack space requires sending unused data in the marshalled stack buffer. Similarly, a round-trip for each system call can add significant latency to input/output intensive operations. Finally, all communication with the syscall proxy server is performed unencrypted. This can be a concern when a remote penetration tester is gaining access to sensitive hosts or files. CORE IMPACT [4], a commercial penetration testing application that employs syscall proxying, addresses these issues by also employing more complex agents with richer network transports that perform encryption. The initial syscall proxy server can be used to deploy a more capable remote agent, but this requires writing an executable to the remote system's hard drive.

Immunity's CANVAS [5], a competing penetration-testing application, uses a custom compiler-based payload construction system called MOSDEF [6]. The MOSDEF compiler takes a subset of the C programming language and generates highly position-independent machine code suitable for dynamic injection into remote processes. The remotely executed payload can make use of system functions or other locally compiled functions. Loops, conditional execution paths, and function calls are all executed inside the remote compromised process and only minimal necessary information is transferred to or from the remote process. This allows the payload author to create complex payloads that can, for example, transfer an entire remote file if successful, or only a smaller error code if an error occurred.

A compiler-style approach such as MOSDEF addresses many of the limitations of syscall proxying at the expense of implementation complexity. A MOSDEF program can create multiple processes and perform a significant amount of work on the client without requiring network round-trips or data transfers. The low-level flexibility of C also allows a MOSDEF program to perform byte-level memory manipulations easily. This is an important capability as performing a heap metadata exploit typically corrupts the heap to the point of causing any subsequent heap operations to crash the program. A MOSDEF program can systematically repair the heap before executing higher-level functions that may require heap memory allocation. MOSDEF, however, does not encrypt the data in transit or payload code. Immunity also provides a remote access trojan called Hydrogen that uses RSA and Twofish to protect communications. The Hydrogen server executable contains an embedded RSA key that is used to securely send the randomly generated

Twofish session key to the client. The Hydrogen server is typically copied to the remote system using MOSDEF and executed [7].

An alternative approach is remote library injection. Remote library injection uses special linking and loading techniques to inject arbitrary shared libraries into a remote compromised process address space. The remote library injection approach has been implemented for both Linux ELF [8] and Windows DLL [9] and both implementations are included in the Metasploit open-source exploit development framework [10]. The benefit of this approach is that arbitrarily complex payloads or existing software can be easily re-linked and remotely executed. The existing implementations load the shared library purely in memory without writing the library to the remote system's hard drive.

The Metasploit project has implemented several payloads using DLL injection, including an injected VNC server [11] for remote graphical control of the host, and the Meterpreter [12], a dynamically extensible server-side scripted environment. The Meterpreter consists of the base Meterpreter library that is remotely injected into the victim process, the interactive shell running on the attacker's host, and custom extensions. Meterpreter extensions add new functions to the Meterpreter shell that are implemented as a combination of local scripting language code and a remotely injected library. The use of a local scripting language (Ruby in Metasploit 3.0 and Perl in prior versions) allows payload functionality to be rapidly developed and deployed on-the-fly. Meterpreter can optionally XOR-encode packets in order to evade Intrusion Detection System signatures.

In a different application but similar spirit, Adam Young and Moti Yung have investigated many ways that strong cryptography can augment and prevent computer viruses [13].

3 Encrypted Payload Protocol

A professional penetration test should not expose the targeted client network to additional security risks in the process of the penetration test, even if the test is being performed across untrusted networks (i.e. the Internet). This includes not sufficiently protecting client data in transit to the penetration tester or using tools that open up additional security vulnerabilities on the client's network during the test. For example, an attacker on the Internet route between a penetration tester and their client target should not be able to piggy-back on the penetration test in order to gain access to the client's network. In addition, the penetration tester may be employing proprietary vulnerabilities, exploits, and techniques that should not be compromised by an unauthorized attacker between the penetration tester and the target network.

We define our attacker to be a passive traffic eavesdropper or active traffic manipulator en route between the penetration tester's systems and the target client network, but not including an adversary on the client's network or a honeypot target. Our penetration test scenario is a client-side penetration test against a client's enterprise network. Client-side penetration tests using web and e-mail based attacks are becoming more common as Internet attackers have also been using these attacks more and more in recent years. In addition, while Internet perimeter security is a relatively mature and understood discipline, client security is still a difficult problem for most organizations.

An open problem with the existing exploit payload systems described above is cryptographically secure payload delivery and communication. Several of the systems described above transmit second-stage agent executables over their first stage payload's communication channels. While the second-stage agents employ strong cryptography to protect communications, they are sent in the clear over the unencrypted first stage payload communication channel. The executable agents are not polymorphic and may easily be fingerprinted and identified by a signature-based intrusion detection system as they are transferred to the target system. Because they are easily identified, an active attacker capable of performing a man-in-the-middle attack could also modify the executables in transit to the target system. The attacker could do this to gain access to the compromised hosts without prior knowledge of the vulnerabilities or exploits used by the penetration tester. Even if the agent executable themselves were made polymorphic, the custom protocols used to deliver them to the target system may still be identified and manipulated by an active attacker.

In a client-side penetration test, exploits are typically delivered as attachments to targeted e-mail messages or on web pages hosted on the penetration tester's web server. The attacks involve some amount of social engineering as the user must be convinced to open the e-mail message, attachment, or web page in order for the exploit to be attempted. Protecting the secrecy of these potentially proprietary exploits in use is a difficult problem that this work does not address. This work does, however, show how establishing a secure channel in the first stage payload protects all communications beyond the delivery of the exploit. This is sufficient to protect against a passive attacker. If the delivery of the exploit can be secured, for example, by sending exploit e-mails using Secure SMTP over TLS or hosting web-based exploits on a SSL web server with a certificate granted by a trusted root certification authority, then the communications between the penetration tester and target network can even be secured against an active attacker able to manipulate traffic. This design provides a secure mechanism for delivering the later-stage exploit payloads, as well as a secure transport for payload and remote agent command and control. This protects the penetration tester's tools, mission logic, and remote target communications from a passive adversary. The polymorphic encoding of the exploit and first-stage payload makes an active attack reasonably difficult, especially so if the exploit is delivered over SSL.

3.1 Architecture

The payload delivery is executed in several stages in order to progressively establish more secure and capable execution environments. In addition, a staged payload system allows the bulk of high-level functionality to be implemented in later stages that typically have less constraints on their implementation. For example, the first stage payload, included in the code injection exploit, typically must be implemented in hand-written assembly in order to guarantee complete position-independent execution. In addition, there may be byte value constraints on the machine code encoding of the payload. This is typically due to the fact that the payload is included with the injection vector within a defined file format or network protocol. The most common example of this constraint is the inability to use the NULL byte in the payload since the value is also used as the ASCII string terminator. To get around this, self-executing payload decoders are typically used to convert an arbitrary machine code fragment into a string consisting of only "safe" byte values. More advanced encoders add polymorphism and restrict the output byte set to printable ASCII, for example.

Our first stage payload initiates communication by negotiating a shared key and establishing a secure channel to the payload delivery server. The secure channel is established by the first stage payload in order to hamper an active adversary desiring to identify and interfere with payload communications. This is done in the first stage because later stage communications may be easier to identify and tamper with than the initial transmission of the actual code injection exploit. For example, consider that browser-based exploits may be delivered over SSL and that exploit payloads typically employ polymorphic self-decoders that may be difficult to identify by signature-based network intrusion detection (IDS) and prevention (IPS) systems. In both cases, identifying and intercepting the delivery of the initial exploit is difficult in principle and impossible using currently available commercial IPS technology. While technically possible, intercepting, decoding, and modifying a polymorphic payload within an arbitrary exploit in transit is a very challenging task.

The secure payload delivery protocol was designed to minimize required code space for the first stage payload

Function	x86 machine code bytes
fp_exptmod_small	135
fp_rshd	173
fp_mulmod_small	80
fp_mul_comba	559
s_fp_sub	336
Total	1283

Figure 1: Cryptographic Routine Code Sizes

through the use of simple yet secure cryptographic operations and algorithms. In addition, custom cryptographic routines were chosen to maximize portability across target operating systems. Our first stage payload establishes the secure channel and must do so given the size constraints of first stage code injection exploit payloads. The system uses ElGamal key agreement (alternatively referred to as Half-Certified Diffie Hellman [14]) to securely establish a session key with the payload server and the RC4 stream cipher to encrypt communications. ElGamal key agreement was chosen because it requires only a single arbitrary-precision integer operation, modular exponentiation. RC4 is also a very simple stream cipher, generating a keystream by swapping elements in its internal S-box. The use of public-key cryptography was chosen over traditional symmetric cryptography in order to prevent a passive attacker from analyzing the initial exploit, recovering the key, and decrypting the subsequent communications. Using public-key cryptography to secure all communication prevents post-analysis by an adversary who has recorded all network traffic.

Both of these cryptosystems were also chosen for their ease of implementation in assembly language or low-level position-independent C. Modular exponentiation, the only mathematical operation required for ElGamal key agreement, can be implemented compactly using classical methods or Montgomery exponentiation[14] for more efficiency. To estimate the code space required for modular exponentiation, we modified an open source fast fixed precision math library [15] to use some more compact algorithms for modular exponentiation and multiplication. The compact modular exponentiation function `fp_exptmod_small` uses right-to-left binary exponentiation[14]. The compact modular multiplication function `fp_mulmod_small` uses repeated subtractions rather than division for the modular reduction step. The total size of the code required to implement modular exponentiation is around 1200 bytes. The exact code sizes of the necessary routines are listed in table 3.1.

The RC4 stream cipher, using a single S-box, may be implemented very compactly. Most stream ciphers are optimized for hardware implementation, but RC4 requires the least number of operations on a general-purpose processor. Additional space required for the use of these cryptographic algorithms is dedicated to the size of the ElGamal public key, roughly twice the bit length of the prime modulus p. Using an elliptic curve cryptography equivalent of ElGamal key agreement would save space for the key at the expense of increased implementation complexity and may actually require more code space. Elliptic-curve cryptography is typically used on devices with low processor power and little memory so the increase in code space in return for smaller keys and thus less computationally intensive operations is a reasonable trade-off. The target systems in our case are assumed to be modern computers with processor power and memory sufficient for basic cryptographic operations. The total payload size, including basic socket networking, ElGamal key agreement, keys, and RC4 routines is estimated to be under two kilobytes, assuming a straightforward x86 assembly language implementation and 1024-bit ElGamal.

The second stage is downloaded over the secure channel created by the first stage and is executed in-place within the vulnerable process' address space. The second stage executes free of code space constraints, but may be executing in a damaged address space. This impediment is overcome by downloading and executing a remote agent executable, described in the next section. Each stage of the payload is described in more detail in the following sections below.

3.2 Stage 1

The Stage 1 payload is the machine code component included in the exploit delivered to the target's web browser or local application. Many injection vectors have limited space available for payload code, so the primary purpose of the Stage 1 payload is to establish communication with a server controlled by the penetration tester in order to download a subsequent stage payload that is free of code size constraints. The payload may be wrapped in a polymorphic decoder in order to evade intrusion detection and eliminate interpreted byte values from the exploit buffer.

The payload includes an ElGamal public key for the payload server (generator g, prime modulus p, public value $x = g^a \mod p$, where a is the secret exponent). The payload proceeds to complete the ElGamal key agreement protocol and compute a session key for use in communicating with the payload delivery server. The payload generates a random number b using an operating-system provided secure random facility. On Windows, `RtlGenRandom` from `ADVAPI32.DLL` can be used to easily generate secure random bytes. Similarly, on Linux or other Unix-like operating systems

```
DecodePayload()
b = Random() mod p
y = g^b mod p
k = x^b mod p
conn = Connect(server)
Send(socket, y)
iv = Read(conn)
RC4Initialize(iv, k)
While(KeepGoing == True)
    length = Read(conn)
    If length == 0
       Then KeepGoing = False
       Break
    End
    RC4Decrypt(k, length)
    code = ReadBytes(conn, length)
    RC4Decrypt(k, code)
    result = Execute(code)
    Send(conn, result)
End
```

Figure 2: Stage 1 Payload Pseudocode

`/dev/urandom` can be read for secure random bytes. The payload then computes $y = g^b \mod p$ using the generator g and prime modulus p from the server's public key. The payload also computes the session key $k = x^b = (g^a)^b = g^{ab}$. Since g is a generator over Z_p^* and b is cryptographically random, the computed session key will have sufficient entropy. The payload proceeds to send y to the payload server and further communication is encrypted using the session key k and the RC4 stream cipher. The initial 256 bytes of RC4 keystream are discarded[16] and separate RC4 keystreams are used for traffic in each direction in order to address weaknesses in RC4. In addition, a random initialization vector is used for each stream to prevent keystream reuse.

Finally, the Stage 1 payload enters a loop repeatedly downloading, decrypting, and executing code from the payload server.

3.3 Stage 2

The Stage 2 payload, executing as position independent machine code fragments within the vulnerable process' address space, is free of the code space constraints of the Stage 1 payload, but there may still be other execution environment constraints. For example, the exploit may have corrupted the heap metadata and subsequent heap operations may cause the process to crash. In these cases, the Stage 2 payload have to repair the heap before attempting to execute more complex operations that require explicit or implicit heap alloca-

```
RepairHeap()
conn = Connect(server)
iv = Read(conn)
RC4Initialize(iv, k)
f = OpenFileForWriting(AGENT)
While(KeepGoing == True)
    length = Read(conn)
    If (length == 0)
        Then KeepGoing = False
        Break
    End
    RC4Decrypt(k, length)
    buffer = ReadBytes(conn, length)
    RC4Decrypt(k, buffer)
    WriteToFile(f, buffer)
End
Execute(AGENT)
```

Figure 3: Stage 2 Payload Pseudocode

tion. Under Windows XP and later Windows operating systems, the default heap can be quickly switched to the low-fragmentation heap using HeapSetInformation(), thus abandoning the use of a potentially corrupted standard default heap.

In order to fully escape execution and implementation constraints, the Stage 2 payload proceeds to download and execute a binary executable from the payload server. In the current design, this executable is a specially-built interpreter for the Lua programming language, described in the next section.

3.4 Limitations

As described above, the payload delivery protocol is not perfectly secure against an active attacker. Many of the design decisions were taken in order to hamper an active attacker, but preventing an active attack is not feasible when the delivery of the exploit is not secure. For instance, an active adversary could modify the first stage payload to insert its own public key in the payload or prevent the generation or use of true random numbers. The only defense against this is the use of polymorphic self-decoders that make identification of the attack more difficult. Alternatively, the penetration tester could host their exploits on a secure web server with a certificate from a trusted root certification authority. This would secure the delivery of the first-stage payload against modification by an active attacker. However, later communication is subject to known-plaintext attacks. The described use of RC4 does not guarantee integrity of the stream. An active attacker with knowledge of the plaintext can choose bytes in the decrypted stream. As described in the threat model above, the system is not truly secure against an active attacker, but takes reasonable effort to make an active attack difficult.

4 Target-Side Scripting

While the common targets of code execution exploits have shifted from servers to client desktops and laptops, most available exploit payloads in commercial and open-source penetration testing tools assume the target remains at a fixed network location. This may not be the case in many modern penetration tests. For example, the easiest way into a network may be through compromising a mobile laptop when it is associated to a “hot spot” wireless network. Existing penetration testing frameworks require that the target be continually reachable by the penetration tester in order to maintain command and control. An exploit payload that supports disconnected operation would allow the penetration tester to take advantage of the target’s mobility in order to gain access to each network that the client connects to.

Several of the existing exploit payload systems offer dynamic scripting support. All of this support, however, is for attacker-side script execution. The execution of these scripts require a constant network connection to the penetration tester’s machine. This may be infeasible when targeting mobile clients such as wireless laptops. In addition, attacker-side logic may require an infeasible amount of input/output to the target. For example, consider a payload with file searching logic. With attacker-side logic, this would require a large amount of uninteresting data to be downloaded to the attacker. Pushing the logic onto the target allows the searching to be performed where the files are local. While attacker-side scripts protect the payload logic, the cryptographic payload delivery system described above adequately protects transport of target-side payload logic to the target. The scripting engine could implement a similar cryptographic protocol or use an available SSL implementation to download scripts from the penetration tester.

The Lua programming language [17] was chosen as the target-side scripting language. Lua has many benefits for this type of application. Its interpreter is small (roughly 200k), very portable by virtue of a pure ANSI C implementation, and the Lua language is quite flexible and powerful. Lua has been historically popular as a scripting and extensibility language for games, however recently many open source security tools have begun to use it as well (Wireshark [18], NMAP [19], and Snort [20]).

The specific Lua interpreter chosen for the target-side scripting environment includes the C/Invoke [21] library, the LuaSocket [22] library, and the custom cryptographic routines used in the previously described secure

```
U32 = clibrary.new("user32.dll",
                   "stdcall")
MessageBox = U32:get_function(Cint,
          "MessageBoxA", Cptr, Cstring,
          Cstring, Cint32)
MessageBox(0, "Hello World", "Lua", 0)
```

Figure 4: Lua C/Invoke Win32 API Example

payloads. This C/Invoke library allows purely scripted code to load and call into arbitrary dynamic library functions. C/Invoke supports marshaling basic types (machine types and strings), structures, and even supports callback functions written in Lua. For these reasons, it was chosen as an ideal dynamic interface to the system APIs. As an example, the Lua programmer would use the code in figure 4 to load the Win32 MessageBox() function and call it.

The LuaSocket library provides interfaces for TCP and UDP sockets as well as higher level protocols like HTTP, SMTP, and FTP. Combined with the included ElGamal key agreement and RC4 encryption routines, LuaSocket's network access can be used to establish secure connections back to the penetration tester for further instructions and to send back encrypted results. At present, the cryptographic support is limited to key agreement and symmetric encryption. This is enough, however, to protect the communication against eavesdropping.

The flexibility offered by the Lua scripting environment with full system library access allows the payload author to rapidly develop and deploy arbitrarily complex payloads. In addition, their implementation as purely textual scripting code allows them to be easily encrypted, signed, downloaded, and stored. Further work will build upon the customized Lua interpreter to build richer cryptographic capabilities.

In the section that follows, we describe several payloads that are difficult or impossible to implement with existing penetration testing tools but straightforward with a target-side scripting payload such as the one described. These payloads may also be implemented as free-standing executables, but there are several benefits to implementing them in a target-side payload scripting environment. As scripts, they are more transient than executables. For example, script code is easier to download and execute on-the-fly whereas an executable must be saved to disk first. This facilitates cleaning up target systems after the penetration testing engagement terminates. The target-side agent could even be configured to automatically remove itself and any downloaded scripts and data at a certain time coinciding with the end of the penetration test.

4.1 Example Payloads

Building rich logic into target-side scripts allows rapid development of payloads that are better suited for client-side penetration tests. These payloads may take advantage of the fact that they may execute without a permanent network connection to the penetration tester. This is useful when attacking mobile clients such as laptops on wireless "hot spot" networks. For example, it may be easiest to compromise an internal network by compromising a laptop with access to that network when the laptop is associated to a "hot spot" wireless network in a coffee shop, airport, or hotel.

An ideal payload for a mobile client would connect back to the penetration tester from whatever network the client was connected to. The payload could monitor the state of the network interfaces, and whenever a network interface became active, it would "phone home" to a server on the Internet. The payload could be configured to automatically terminate at a certain date, coinciding with the end of the penetration testing engagement.

A second payload style that takes advantage of the target-side logic is the "file searcher" payload. This payload would search local documents and files on the target systems' hard drives for key words or patterns. Matching files would be collected, encrypted, and sent to the penetration tester for analysis. Performing the file searching on the target prevents excessive amounts of file data from being sent across the network.

Finally, remote target-side payloads could scan the remote network for other vulnerabilities or automatically gain access to other remote systems. For example, a long running password brute force may be unreasonable to launch when the penetration tester must be continually connected to the target system. If the process were run autonomously on a compromised system, the attack could proceed disconnected and report results asynchronously. A similar model would be beneficial for other long-running attacks such as network sniffing and hosting web exploits on the target network. Hosting web exploits on the target's internal network typically grants the attacker some level of privilege escalation as the exploits are placed within Internet Explorer's Local Intranet security zone thus gaining additional privileges over an Internet-based web attack.

5 Conclusion

The payload described above established an encrypted channel that is secure against passive eavesdropping during or after the attack. It is not, however, perfectly secure against a malicious user able to perform a man-in-the-middle attack because the public key of the payload delivery server is included in the exploit. However, an attacker taking advantage of this must be able to actively identify and intercept the exploit as it is being delivered to the target. This is somewhat unlikely as it requires an exact signature for the specific exploit being used or a means to identify, simulate, and replace polymorphic self-decoding executable code.

Assuming that the exploit delivery has not been tampered with, the delivery and communication of the target-side scripting system is secure against both eavesdropping and man-in-the-middle attacks. An active attacker with knowledge of the plaintext may, however, modify those bytes within the RC4 encrypted streams. The delivered executable does nothing to protect itself against recovery from the target filesystem. This is typically not an issue in professional penetration testing and intentionally makes the payload unsuitable for illicit activities. The mission-logic encapsulated in Lua script, however, is never written to disk and is secure against eavesdropping and filesystem recovery.

Future work will enrich the capabilities of the system to perform in-memory Lua interpreter injection and implement richer cryptographic support within the Lua interpreter. In addition, research into secure asynchronous command and control protocols would provide an ideal remote management system for deployed payloads. The author believes that penetration-testing techniques and tools must grow to resemble the Internet attacker technology used in drive-by downloads, botnets, and e-mail attacks in order to better evaluate an organization's defenses against these threats.

References

[1] P.-H. Kamp and R. N. M. Watson, "Jails: Confining the omnipotent root," in *Proceedings of the 2nd International SANE Conference*, 2000.

[2] K. J. Biba, "Integrity considerations for secure computer systems," Tech. Rep. MTR-3153, The Mitre Corporation, April 1977.

[3] M. Caceres, "Syscall proxying - simulating remote execution." CORE Security Technical Report, 2002.

[4] CORE Security Technologies, "Impact." http://www.coresecurity.com/products/coreimpact/.

[5] Immunity Security, "Canvas." http://www.immunitysec.com/products-canvas.shtml.

[6] D. Aitel, "Mosdef," in *BlackHat Briefings Federal*, 2003.

[7] D. Aitel. personal communication, July 2007.

[8] A. E. Cuttergo, "The joys of impurity." http://archives.neohapsis.com/archives/vuln-dev/2003-q4/0006.html, October 2003.

[9] M. Miller and J. Turkulainen, "Remote library injection." http://www.nologin.net/Downloads/Papers/remote-library-injection.pdf.

[10] Metasploit Project, "Metasploit framework." http://www.metasploit.org.

[11] AT&T Laboratories Cambridge, "Virtual network computing." http://www.cl.cam.ac.uk/research/dtg/attarchive/vnc/.

[12] M. Miller, "Metasploit's meterpreter." http://www.nologin.org/Downloads/Papers/ meterpreter.pdf, December 2004.

[13] A. Young and M. Yung, *Malicious Cryptography: Exposing Cryptovirology*. Wiley, 2004.

[14] S. A. V. Alfred J. Menezes, Paul C. van Oorschot, *Handbook of Applied Cryptography*. CRC, 1996.

[15] T. S. Denis, "Tomsfastmath." http://libtom.org.

[16] S. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the key scheduling algorithm of RC4," *Eighth Annual Workshop on Selected Areas in Cryptography*, August 2001.

[17] L. H. d. F. Roberto Lerusalimschy, Waldemar Celes, "The programming language lua." http://www.lua.org.

[18] G. Combs, "Wireshark network analyzer." http://www.wireshark.org.

[19] Fyodor, "Nmap security scanner." http://www.insecure.org/nmap/.

[20] M. Roesch, "Snort network intrusion detection system." http://www.snort.org.

[21] W. Weisser, "C/invoke." http://www.nongnu.org/cinvoke/.

[22] D. Nehab, "Luasocket: Network support for the lua language." http://www.cs.princeton.edu/ diego/professional/luasocket/.