

Midterm Update: Correlating Multi-Session Attacks with Replay

1 Introduction

Non-deterministic application replay has the potential to be a powerful tool in intrusion analysis and recovery. Unlike deterministic replay, it allows forensic investigators to ask “what if” questions based on log data and to reason about the sources and methods of intrusion. It also poses many new challenges, not the least of which is responding to the changes in the behaviour of applications as a result of non-determinism. For example, what is the appropriate response to a network message which is “slightly” different during replay, or a request for external data beyond that which was originally logged? Analysis of different non-deterministic network replay strategies has convinced me that performing network replay reliably is a very difficult problem and that achieving any measure of reliability is predicated on constraining the requirements to include reliability only for specific protocols and sometimes, specific use patterns.

2 Status of Work

Up to this point, most of the work has been concentrated on solving the challenges and corner cases that can occur from replaying various protocols. Toward this end, several small traces of protocol data have been captured and some simple experiments have been performed. The following sections detail the major classes of problems that complicate network replay. The implementation of the design is currently in progress.

3 Issues

It would be un-wise to assume that an application under replay would produce exactly the same requests as it did during first execution. Time stamps, IP addresses, random data such as transaction IDs and other “minor” protocol fields will change and in most cases it will be acceptable to replay the stored network data. Therefore any attempt at network replay cannot rely on the application producing exactly the same network activity and needs to be sufficiently adaptable or tolerant in the presence of such changes.

Because network replay is hard, one may be tempted to allow applications under replay to access the live servers, avoiding the intricacies of automated network replay. However, allowing applications under replay to access the real network would be troublesome when the remote servers are not under the same administrative control and therefore may be unavailable or their state has changed. Additionally, for server applications there would

be no obvious way of replaying client activity. Therefore, automatic network replay is indeed necessary in many cases. The follows subsections details some of the major issues encountered during the design of the system.

3.1 Coalescing Application Data Units

The simplest approach to network replay is to record the incoming data stream for each remote IP/Port pair and allow applications to read from these streams during replay without keeping any additional information or state. This approach runs into problems because the network replay module does not know how much data to return for each request. A socket read may request a different amount during replay and it is possible that allowing the application to read its requested number of bytes from the stored stream would cause it to read multiple or fractional Application Data Units (ADUs).

Example: During original execution, a web browser requests a web page and passes in an empty 100 byte receive buffer when it attempts to read from a socket. The web server returns a 50 byte web page. The user presses refresh and another 100 byte read occurs which is again satisfied with a 50 byte web page. If the web browser is allowed to read its requested 100 bytes from the stored stream during replay, it will receive two web pages for one request.

Solution: It is necessary to perform ADU fragmentation and to feed the fragments back to the application one at a time to prevent coalescing of ADUs. Writing code to perform fragmentation for every possible protocol is error-prone and unscalable. It may be possible to infer fragmentation by recording the length of fragments originally returned to the application. Additionally, coalescing will not always be a problem, such as in media streaming protocols.

3.2 Identifying Request/Response Pairs

Even with the adoption of ADU fragmentation, there are still situations where too much data may be returned to a read request.

Example: During original execution, a telnet client connects to a remote machine. The session is interactive; a burst of data from the client (for example, a command) will prompt a burst of data from the telnet server (command output). If the network replay module does not recognize that application sends cause application reads and satisfies every read request immediately, it will allow the application to read the entire stream before the application has a chance to send out any data. This would manifest itself as the entire output of the session being displayed on screen before the client issues any commands.

Solution: For most protocols that we are interested in, it makes sense to track the pattern of requests and responses that occur on the same connection and to assume that sent data

produces received data. This means that the replay module will not allow the application to read the recorded data stream until it has seen a request (a send) occur assuming that a request was present during original execution. Clearly, this approach will not work for every protocol and it is impossible to reliably deduce causality based on timing but it should be satisfactory for many protocols.

3.3 Correcting Cookie Fields

It is possible to solve the previous two problems for most protocols without resorting to application-specific knowledge. There are cases, however, where it is necessary to rewrite the headers of certain protocols. For example, if the activity of a Web server is being replayed, and the stored network data is the activity of clients connecting to the web server, it is necessary for the client HTTP requests to be rewritten such that their HTTP headers contain the new cookie values set by the server. Therefore, it is necessary to introduce protocol-specific code into the replay module. This may be acceptable since it does not require much work and can be done for only for a few protocols.

3.4 Masking Failures

Network replay may unwittingly disrupt the process of intrusion analysis.

Example: Consider the replay of an un-authorized login into a remote file server. The attacker, John, copies Mike's login key into his home directory. He then proceeds to access the remote server using Mike's credentials. During replay, the investigator removes the login key and observes the behaviour of the system. This time the application tries to log in as John. Because network replay will reproduce the same network activity as last time, it will make it appear as if John is still able to log into the remote server despite not having the correct login key. Since the resulting behaviour is essentially the same, the investigator may be misled into believing the presence of the login key was not essential for the attack.

Solution: This is an inherent limitation of using stored network data, therefore this problem needs to be solved by the investigator, possibly by allowing the application to access the real network for certain types of connections.

4 Conclusion

At this point, obstacles to implementation have been identified and resolved, and the scope of the design has been refined. It is difficult to estimate the effectiveness of the solutions described here but they should work well enough in many cases and it is appropriate to proceed with implementation before refining the approach any further.