Protocol-Independent Adaptive Replay of Application Dialog

Weidong Cui[†], Vern Paxson[‡], Nicholas C. Weaver[‡], Randy H. Katz[†]
[†]University of California, Berkeley, CA
[‡]International Computer Science Institute, Berkeley, CA

Abstract

For many applications—including recognizing malware variants, determining the range of system versions vulnerable to a given attack, testing defense mechanisms, and filtering multi-step attacks—it can be highly useful to mimic an existing system while interacting with a live host on the network. We present RolePlayer, a system which, given examples of an application session, can mimic both the client side and the server side of the session for a wide variety of application protocols. A key property of RolePlayer is that it operates in an application-independent fashion: the system does not require any specifics about the particular application it mimics. It instead uses byte-stream alignment algorithms to compare different instances of a session to determine which fields it must change to successfully replay one side of the session. Drawing only on knowledge of a few low-level syntactic conventions (such as representing IP addresses using "dotted quads"), and contextual information such as the domain names of the participating hosts, RolePlayer can heuristically detect and adjust network addresses, ports, cookies, and length fields embedded within the session, including sessions that span multiple, concurrent connections on dynamically assigned ports.

We have successfully used RolePlayer to replay both the client and server sides for a variety of network applications, including NFS, FTP, and CIFS/SMB file transfers, as well as the multi-stage infection processes of the Blaster *and* W32.Randex.D *worms.*

1 Introduction

In a number of different situations it would be highly useful if we could cheaply "replay" one side of an application session in a slightly different context. For example, consider the problem of receiving probes from a remote host and attempting to determine whether the probes reflect a new type of malware or an already known attack. Different attacks that exploit the same vulnerability often conduct the same application dialog (e.g., the many steps of a Windows file-access session) prior to finally exposing their unique malicious intent. Thus, to coax from these sources their final intent requires engaging them in an initial dialog, either by implementing protocol-specific application-level responders [15, 16] or by deploying high-interaction honeypot systems [6, 19] that run the actual vulnerable services. Both approaches are expensive in terms of development or management overhead.

On the other hand, much of the dialog required to engage with the remote source follows the same "script" as seen in the past, with only very minor variants (different hostnames, IP addresses, port numbers, command strings, or session cookies). If we could cheaply reproduce one side of the dialog, we could directly tease out the remote source's intent by efficiently driving it through the routine part of the dialog until it reaches the point where, if it is indeed something new, it will reveal its distinguishing nature by parting from the script.

Another example comes from trying to determine the equivalent for malware of a "toxicology spread" for a biological pathogen. That is, given only an observed instance of a successful attack against a given type of server (i.e., particular OS version and server patch level), how can we determine what other server/OS versions are also susceptible? If we have instances of other possible versions available, then we could test their vulnerability by replaying the original attack against them, providing the replay again takes into account the natural session variants such as differing hostnames, IP addresses, session cookies, etc. Similarly, we could use replay to feed possible attacks into more sophisticated analysis engines (e.g., [14]).

We can also use lightweight replay to facilitate testing of network systems. For example, when developing or configuring a new security mechanism it can be very helpful if we can easily repeat attacks against the system to evaluate its response. Historically, this can require a complex testbed to repeatedly run a piece of malware in a safe, restricted fashion. Armed with a replay system, however, we could capture a single instance of an attack and then replay it against refinements of the security mechanism without having to reestablish an environment for the malware to execute within.

We can generalize such repeated replay to tasks of stress-testing, evaluating servers, or conducting large-scale measurements. A replay system could work as a low-cost client by replaying application dialogs with altered parameters in part of the dialog. For example, by dynamically replacing the receiver's email address in a SMTP dialog, we could use replay to create numerous SMTP clients that send the same email to different addresses. We could use such a capability for both debugging and performance testing, without needing to either create specialized clients or invoke repeated instances of a computationally expensive piece of client software.

A final, powerful example concerns the use of replay to construct *proxies*. Suppose in the first example above that not only do we wish to determine whether an incoming probe reflects a new type of attack or a known type, but we also want to *filter* the attack if it is a known type but allow it through if it is a new type. We could use replay to efficiently do so in two steps. First, we would replay the targeted server's behavior in response to the probe's initial activity (e.g., setting up a Windows SMB RPC call) until we reach a point where a new attack variant will manifest itself (e.g., by making a new type of SMB call or by sending over a previously unseen payload for an existing type of call). If the remote host at this point proves to lack novelty (we see the same final step in its activity as we have seen before), then we drop the connection. However, if it reveals a novel next step, then at this point we would like it to engage a high-interaction honeypot server so we can examine the new attack. To do so, though, we must bring the server "up to speed" with respect to the application dialog in which the remote host is already engaged. We can do so by using replay *again*, this time replaying the remote host's previous actions to the new server so that the two systems become synchronized, after which we can allow the remote host to proceed with its probing. If we perform such proxying correctly, the remote will never know that it was switched from one type of responder (our initial replayer) to another (the high-interaction honeypot).

In this paper we develop a system, RolePlayer, to provide such replay functionality. We aim for the system to achieve several important goals:

- **Protocol independence**. The system should not need any application-specific customization, so that it works transparently for a large class of applications, including both as a client and as a server.
- **Minimal training**. The system should be able to mimic a previously seen type of application dialog given only a small number of examples.
- **Automation**. Given such examples, the system should operate correctly without requiring any manual intervention.

In some cases, replaying is trivial to implement. For example, each attack by the Code Red worm sends exactly the same byte stream over the network to a target. However, successfully replaying an application dialog can be much more complicated than simply parroting the stream. Consider for example the Blaster worm of August, 2003, which exploited a DCOM RPC vulnerability in Windows by attacking port 135/tcp. For Blaster, if a compromised host (A) finds a new vulnerable host (V) via its random scanning, then the following infection process occurs.

1. A opens a connection to 135/tcp on V and sends three packets with payload sizes of 72, 1460, and 244 bytes. These packets compromise V and open a shell listening on 4444/tcp.
2. A opens a connection to 4444/tcp and issues "tftp -i xx.xx.xx.xx GET msblast.exe" where "xx.xx.xx.xx" is A's IP address.
3. V sends a request back to 69/udp on A to download msblast.exe via TFTP.
4. A issues commands via 4444/tcp to start msblast.exe on host V.

This example illustrates a number of challenges for implementing replay.

1. An application session can involve multiple connections, with the initiator and responder switching roles as both client and server, which means RolePlayer must be able to run as client and server simultaneously.
2. While replaying an application dialog we sometimes cannot coalesce data. For example, if the replayer attempts to send the first Blaster data unit as a single packet, we observe two packets with sizes of 1460 and 316 bytes due to Ethernet framing. For reasons we have been unable to determine, these packets do not compromise vulnerable hosts. Thus, RolePlayer must consider both application data units and network framing. (It appears that there is a race condition in Blaster's exploitation process—A connects to 4444/tcp before the port is opened on V. If we do not consider the network framing, it increases the likelihood of the race condition. Accommodating such timing issues in replay remains for future work.)
3. Endpoint addresses such as IP addresses, port numbers and hostnames may appear in application data. For example, the IP address of A appears in the TFTP download command. This requires RolePlayer to find and update endpoint addresses dynamically.

4. Endpoint addresses (especially names but also IP addresses, depending on formatting) can have variable lengths, and thus the data size of a packet or application data unit can differ between dialogs. This creates two requirements for RolePlayer: deciding if a received application data unit is complete, particularly when its size is smaller than expected; and changing the value of such length fields when replaying them with different endpoint addresses.

5. Some applications use "cookie" fields to record session state. For example, the process ID of the client program is a cookie field for the Windows CIFS protocol, while the file handle is one in the NFS protocol. Therefore, RolePlayer must locate and update these fields during replay.

6. We observe that non-zero padding up to 3 bytes may be inserted into an application data unit, which we must accommodate without confusion during replay.

We organize the remainder of the paper as follows. We compare RolePlayer with previous work and highlight its contributions in Section 2. After presenting terminology and our design assumptions in Section 3, we describe the design of RolePlayer in detail in Section 4. We discuss our evaluation methodology and results in Section 5, and summarize in Section 6.

2 Related work

The programming and operating systems communities have studied the notion of replaying program execution for a number of years [1, 5, 7, 9, 17, 18]. However, we know of little literature discussing general replay of network activity at the application level. The existing work has instead focused on incorporating application-specific semantics [15, 16].

Libes' *expect* tool includes the same notion as our work of automatically following a "script" of expected interactive dialogs [10]. A significant difference of our work, however, is that we focus on generating the script automatically from previous communications, and we have not yet incorporated the possibility of the script including branch-points that lead to alternatives in the dialog.

A number of commercial security products address replaying network traffic [4, 12]. These products however appear limited to replay at the network or transport layer, similar to the *Monkey* and *Tcpreplay* tools [3, 20]. The *Flowreplay* tool uses application-specific plug-ins to support application-level replay [21].

Our work leverages the Needleman-Wunsch algorithm [13], widely used in bioinformatics research, to locate fields that have changed between one example of an application session and another. In this regard, our approach is similar to the recent *Protocol Informatics* project [2], which attempts to identify protocol fields in unknown or poorly documented network protocols by comparing a series of samples using sequence alignment algorithms.

Compared to the previous work, RolePlayer makes four contributions:

1. It does not require knowledge of application-level protocol semantics.

2. It automatically generates its dialog "script" from previously recorded traffic.

3. It can replay against live peers as both initiator and responder because it can adapt to dynamic changes such as session identifiers and differing hostnames.

4. It requires at most two samples to replay a single particular application session.

3 Terminology and design assumptions

We will use the term *application session* to mean a fixed series of interactions between two hosts that accomplishes a specific task (e.g., uploading a particular file into a particular location). The term *application dialog* refers to a recorded instance of a particular application session. The host that starts a session is the *initiator*. The initiator contacts the *responder*. (We avoid "client" and "server" here because for some applications the two endpoints assume both roles at different times.) We want RolePlayer to be able to correctly mimic both initiators and responders. In doing so, it acts as the *replayer*, using previous dialog(s) as a guide in communicating with the remote *live peer*.

An application session consists of a set of TCP and/or UDP *connections*, where a UDP "connection" is a pair of unidirectional UDP flows. In a connection, an *application data unit* (ADU) is a consecutive chunk of application-level data sent in one direction, which spans one or more packets. RolePlayer only cares about application-level data, ignoring both network and transport-layer headers when replaying a session.

Within an ADU, a *field* is a byte sequence with semantic meaning. A *dynamic field* is a field that potentially changes between different dialogs. We classify dynamic fields into five types: host-specific *endpoint-address* (e.g., hostnames, IP addresses, transport port numbers), *length* (1 or 2 bytes reflecting the length of either the ADU or a subsequent dynamic field), *cookie* (session-specific opaque data that appears in ADUs from both sides of the dialog, such as transaction IDs), *argument* (fields that customize the meaning of a session, such as the destination directory for a file transfer, or the domain name in a DNS lookup), and *don't-care* (opaque fields that appear in only one side of the dialog). Argument fields are only relevant for replay if we want to specifically alter them; for *don't-care* fields, we ignore the difference if the value for them we receive from a live peer differs from the original, and we send them verbatim if communicating them to a live peer.

To replay a session, we need at least one sample dialog to use as a reference, the *primary* application dialog. We may also need an additional *secondary* dialog for discovering dynamic fields, particularly length fields. Finally, we refer to the dialog generated during replay as the *replay* dialog.

Given this terminology, we can frame our design assumptions as follows.

1. We have available primary and (when needed) secondary dialogs that differ enough to disclose the dynamic fields, but are otherwise the same in terms of the application semantics.

2. We assume that the live peer is configured suitably similar to its counterpart in the dialogs.

3. We assume some standard network protocol representations, such as embedded IP addresses being represented either as a four-byte integer or in dotted or comma-separated format and embedded transport port numbers as two-byte integers. (We currently consider network byte order only, but it is an easy extension to accommodate either-endian.) Note that we do *not* assume the use of a single format, but instead encode into RolePlayer each of the possible formats.

4. We assume the domain names of the participating hosts in sample dialogs can be provided by the user if required for successful replay.

5. We assume that the application session does not include time-related behavior (e.g., wait 30 seconds before sending the next message) and does not require encryption or cryptographic authentication.

4 RolePlayer design

The basic idea of RolePlayer is straightforward: given an example or two of an application session, locate the dynamic fields in the ADUs and adjust them as necessary before sending the ADUs out from the replayer. Since some dynamic fields, such as length fields, can only be found by comparing two existing sample application dialogs, we split the work of RolePlayer into two stages: *preparation* and *replay*. During preparation, RolePlayer first searches for endpoint-address and argument fields in each sample dialog, then searches for length fields and possible cookie fields by comparing the primary and secondary dialogs. During replay, it first searches for new values of dynamic fields by comparing received ADUs with the corresponding ones in the primary dialog, then updates them with the new values. In this section, we describe both stages in detail.

Before proceeding, we note that a particularly important issue concerns dynamic ports. RolePlayer needs to determine when to initiate or accept new connections, and in particular must recognize additional ports that an application protocol dynamically specifies. To do so, RolePlayer detects stand-alone ports and hostname/port pairs during its search process, and matches these with subsequent connection requests. This enables the system to accommodate features such as FTP's use of dynamic ports for data transfers, and portmapping as used by SunRPC.

4.1 The preparation stage

In the preparation stage, RolePlayer needs to parse network traces of the application dialogs and search for the dynamic fields within. For its processing we may also need to inform RolePlayer of the hostnames of both sides of the dialog, and any application arguments of interest, as these cannot be inferred from the dialog.

4.1.1 Parsing application dialogs

RolePlayer organizes dialogs in terms of both ADUs and data packets. It uses data packets as the unit for sending and receiving data and ADUs as the unit for manipulating dynamic fields. Note that data packets may interleave with ADUs. For example, FTP sessions use two concurrent connections, one for data transfer and one for control. RolePlayer needs to honor the ordering between these as revealed by the packet sequencing on the wire, such as ensuring that a file transfer on the data channel completes before sending the "226 Transfer complete" on the control channel. Accordingly, we use the SYN, FIN and RST bits in the TCP header to delimit the beginning and end of each connection, so we know the correct temporal ordering of the connections within a session.

4.1.2 Searching for dynamic fields

RolePlayer next searches for dynamic fields in the primary dialog, and also by comparing it with the secondary dialog (if available). The searching process contains a number of subtle steps and considerations. We illustrate these using replay of a fictitious toy protocol, SPD (Service Port Discovery; see Appendix A) as a running example. SPD consists of a single client request carrying the client's hostname and a service name, to which the server replies with an IP address and port number expressed in the comma-separated syntax used by FTP. We note that this protocol is sufficiently

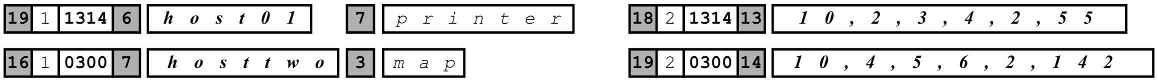


Figure 1. The captured primary and secondary dialogs for requests (left) and responses (right) using the toy SPD protocol. RolePlayer first discovers endpoint-address (bold italic) and argument (italic) fields, then breaks the ADUs into segments (indicated by gaps), and discovers length (gray background) and possible cookie (bold) fields.

simple that some of the operations we illustrate appear trivial. However, the key point is that the same operations also work for much more complex protocols (such as Windows SMB/CIFS).

Figure 1 shows two SPD dialogs, each consisting of a request (left) and response (right). For RolePlayer to process these, we must inform it of the embedded hostnames ("host01", "hosttwo") and arguments ("printer", "map"), though we do not need to specify their locations in the protocol. If we did not specify the arguments, they would instead be identified and treated as *don't-care* fields. We do not need to inform RolePlayer of the hostname of a live peer because RolePlayer can automatically find it if it appears in an ADU from the live peer.

In addition, we do *not* inform RolePlayer of the embedded transaction identifiers (1314, 0300), length fields, IP addresses (10.2.3.4, 10.4.5.6), or port numbers ($567 = 2 \cdot 256 + 55$, $654 = 2 \cdot 256 + 142$).

The naive way to search for dynamic fields would be to align the byte sequences of the corresponding ADUs and look for subsequences that differ. However, we need to treat endpoint-address and argument fields as a whole; for example, we do *not* want to decide that one difference between the primary and secondary dialogs is changing "01" in the first to "two" in the second (the tail end of the hostnames). Similarly, we want to detect that **13** and **14** in the replies are length fields and not elements of the embedded IP addresses that changed. To do so, we proceed as follows:

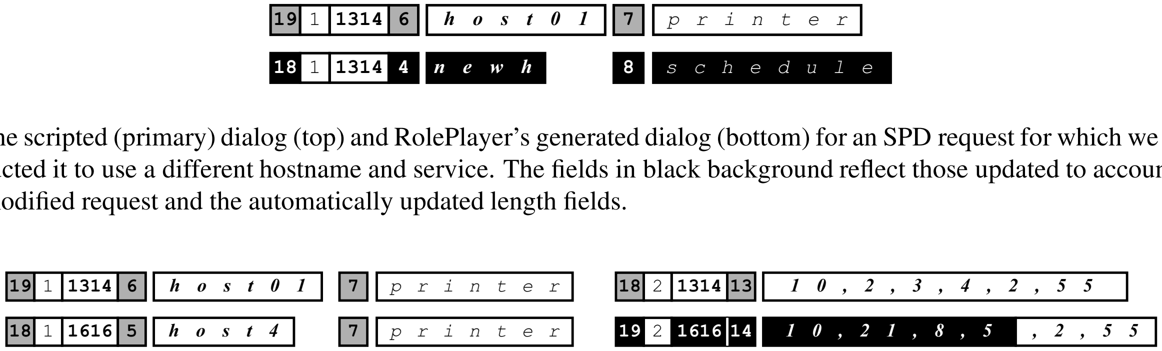
1. Search for endpoint-address and argument fields in both dialogs by finding matches of *presentations* of their known values. For example, we will find "host01" as an endpoint-address field and "printer" as an argument field in the primary's request.

   We consider seven possible presentations and their Unicode [22] equivalents for endpoint addresses, and one presentation and its Unicode equivalent for arguments. For example, for the primary's reply we may know from the packet headers in the primary trace that the server's IP address is 10.2.3.4, in which case we search for: the binary equivalent (0x0A020304); ASCII dotted-quad notation ("10.2.3.4"); and comma-separated octets ("10,2,3,4"). The latter locates the occurrence of the address in the reply. (If the server's address was something different, then we would *not* replace "10,2,3,4" in the replayed dialog.)

2. If we have a secondary dialog, then RolePlayer splits each ADU into segments based on the endpoint-address and argument fields found in the previous step. It is possible that endpoint-address and argument fields do not match due to some bogus matches of the presentations found in *don't-care* fields. We remove these bogus fields before splitting each ADU.

3. Finally, RolePlayer searches for length, cookie, and *don't-care* fields by aligning and comparing each pair of data segments. By "alignment" here we mean application of the Needleman-Wunsch algorithm [13], which efficiently finds the minimal set of difference between two byte sequences subject to constraints; see Section 4.3.1 below for discussion. An important point is that at this stage we do not distinguish between cookie fields and *don't-care* fields. Only during the actual subsequent live session will we see whether these fields are used in a manner consistent with cookies (which need to be altered during replay) or *don't-care*'s (which shouldn't). See Section 4.2 for the process by which we make this decision.

In the SPD example, aligning and comparing the five-byte initial segments in the primary and secondary requests results in the discovery of two pairs of length fields (19 vs. 16, and 6 vs. 7) and one cookie field (1314 vs. 0300). To find these, RolePlayer first checks if a pair of differing bytes (or differing pairs of bytes) are consistent with being length fields, i.e., their numeric values differ by one of: (1) the length difference of the whole ADU, (2) the length difference of an endpoint-address or argument fields that comes right after the length fields, or (3) the double-byte length difference of these, if the subsequent field is in Unicode format. For example, RolePlayer finds 19 and 16 as length fields because their difference matches the length difference of the request messages, while the difference between 6 and 7 matches the length difference of the client hostnames. (Note that, to accommodate Unicode, we merge



(a) The scripted (primary) dialog (top) and RolePlayer's generated dialog (bottom) for an SPD request for which we have instructed it to use a different hostname and service. The fields in black background reflect those updated to account for the modified request and the automatically updated length fields.

(b) The same for constructing an SPD reply to a live peer that sends a different transaction ID in their request, and for which the replayer is running on a different IP address. Note that the port in the reply stays constant.

Figure 2. Initiator-based and responder-based SPD replay dialogs.

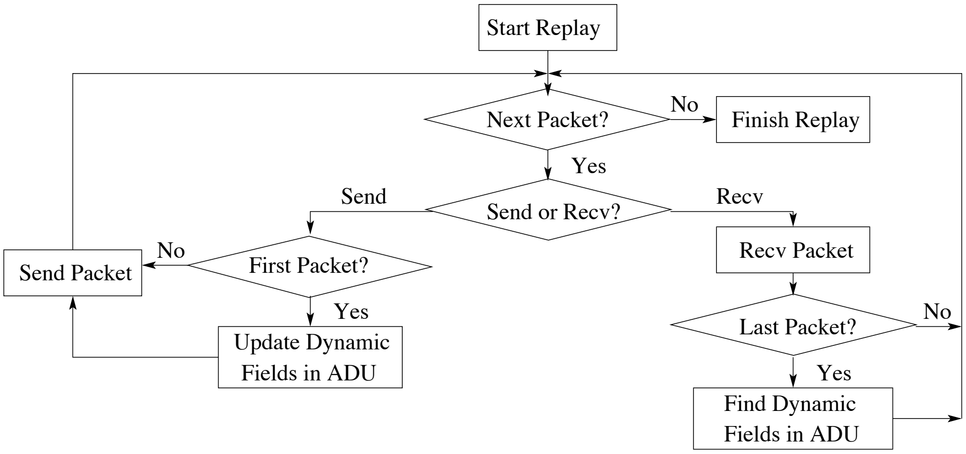


Figure 3. Steps in the replay process.

any two consecutive differing byte sequences if there is only one single zero byte between them.)

4.2 The replay stage

With the preparation complete, RolePlayer can communicate with a live peer. To do so, it uses the primary application dialog plus the discovered dynamic fields as its "script," allowing it to replay either the initiator or the responder. Figures 2(a) and 2(b) give examples of creating an initial request and responding to a request from a live peer in SPD, respectively. To construct these requires several steps, as shown in Figure 3.

4.2.1 Deciding packet direction

We read the next data packet from the script to see whether we now expect to receive a packet from the live peer or send one out.

4.2.2 Receiving data

If we expect to receive a packet, we read data from the specific connection. Doing so requires deciding whether the received data is complete (i.e., equivalent to the corresponding data in the script). To do so, we use the alignment algorithm (Section 4.3.1) with the constraint that the match should be weighted to begin at the same point (i.e., at the beginning of the received data). If it yields a match with no trailing "gap" (i.e., it did not need to pad the received data to construct a good match), then we consider that we have received the expected data. Otherwise, we wait for more data to arrive.

After receiving a complete ADU, we compare it with the corresponding one in the script to locate dynamic fields. This additional search is necessary for two reasons. First, RolePlayer may need to find new values of endpoint addresses or arguments. Second, cookie fields found in the replay stage may differ from those found in the preparation stage due to accidental agreement between the primary and secondary dialogs.

We can apply the techniques used in the preparation stage to find dynamic fields, but with the major additional challenge that now for the live dialog we do *not* have access to the endpoint addresses and application arguments. While the script provides us with guidance as to the existence of these fields, they are often not in easily located positions but instead surrounded by *don't-care* fields. The difficult task is to pinpoint the fields that need to change in the midst of those that do not matter. If by mistake this process overlaps an endpoint-address or argument field with a *don't-care* field, then this will likely substitute incorrect text for the replay. However, we can overcome these difficulties by applying the alignment algorithm (Section 4.3.1) with pairwise constraints. Doing so, we find that RolePlayer correctly identifies the fields with high reliability.

After finding the endpoint-address and argument fields, we then use the same approach as for the preparation stage to find length, cookie and *don't-care* fields. We save the data corresponding to newly found endpoint-address, argument, and cookie fields for future use in updating dynamic fields.

4.2.3 Sending data

When the script calls for us to send a data packet, we check whether it is the first one in an ADU. If so, we update any dynamic fields in it and packetize it. The updated values come from three sources: (1) analysis of previous ADUs; (2) the IP address and transport port numbers on the replayer; (3) user-specified (for argument fields). After updating all other fields, we then adjust length fields.

This still leaves the cookie fields for updating. RolePlayer only updates cookie fields *passively*, i.e., to reflect changes first introduced by the live peer. So, for example, in Figure 2(a) we do not change the transaction ID when sending out the request, but we do change it in the reply shown in Figure 2(b).

To update cookie values altered by the live peer, we search the ADU we are currently constructing for matches to cookie fields we previously found by comparing received ADUs with the script. However, some of these identified cookie fields may in fact not be true cookies (and thus should not be reflected in our new ADU), for three reasons: some Windows applications use non-zero-byte padding; sometimes a single, long field becomes split into multiple, short cookie fields due to partial matches within it; and messages such as FTP server greetings can contain inconsistent (varying) data.

Thus, another major challenge is to robustly determine which cookie matches we should actually update. We studied several popular protocols and found that cookie fields usually appear in the same context. Also, the probability of a false match to a N-byte cookie field is very small when N is large (e.g., when $N \geq 4$). Hence, to determine if we should update a cookie match, we check four conditions, requiring at least two to hold.

1. *Is the byte sequence ≥ 4 bytes?* This condition captures the fact that padding fields are usually < 4 bytes in length because they are used to align an ADU at a 32-bit boundary.

2. *Does the byte sequence overlap a potential cookie field found in the preparation stage?* (If there is no secondary dialog, this condition will always be false, because we only find cookie fields during preparation by comparing the primary dialog with the secondary dialog.) The intuition here is that the matched byte sequence is more likely to be a correct one since it is part of a potential cookie field.

3. *Is the prefix of the byte sequence consistent with that of the matched cookie field?* The prefix is the byte sequence between the matched cookie field and the preceding non-cookie dynamic field (or the beginning of the ADU). For prefixes exceeding 4 bytes, we consider only the last four bytes (next to the targeted byte sequence). For empty prefixes, if the non-cookie dynamic fields are the same type (or it is the beginning of the ADU) then the condition holds. This condition matches cookie fields with the same leading context.

4. *Is the suffix consistent?* The same as the prefix condition but for the trailing context.

4.3 Design issues

4.3.1 Sequence alignment

The cornerstone of our approach is to compare two byte streams (either a primary dialog and a secondary dialog, or a script and the ADUs we receive from a live peer) to find the best description of their differences. The whole trick here is what constitutes "best." Because we strive for an application-independent approach, we cannot use the semantics of the underlying protocol to guide the matching process. Instead we turn to generic algorithms that compare two byte streams using customizable, byte-level weightings for determining the significance of differences between the two.

The term used for the application of these algorithms is "alignment," since the goal is to find which subsequences within two byte streams should be considered as counterparts. The Needleman-Wunsch algorithm [13] we use is parameterized in terms of weights reflecting the value associated with identical characters, differing characters, and missing characters ("gaps"). The algorithm then uses dynamic programming to find an alignment between two byte streams (i.e., where to introduce, remove, or transform characters) with maximal weight.

We use two different forms of sequence alignment, global and semi-global. Global refers to matching two byte streams against one another in their entirety, and is done using the standard Needleman-Wunsch algorithm. Semi-global reflects matching one byte stream as a prefix or suffix of the other, for which we use a modified Needleman-Wunsch algorithm. (Due to space limitations we do not present algorithmic details here.)

When considering possible alignments, the algorithm assigns different weightings for each aligned pair of characters depending on whether the characters agree, disagree, or one is a gap. Let the weight be m if they agree, n if they disagree, and g if one is a gap. The total score for a possible alignment is then the sum of the corresponding weights. For example, given *abcdf* and *acef*, with weights $m = 2, n = -1, g = -2$, the optimal alignment (which the algorithm is guaranteed to find) is *abcdf* with *a-cef*, with score $m + g + m + n + m = 3$, where – indicates a gap.

To compute semi-global alignments of matching one byte stream as prefix of the other, we modify the algorithm to ignore trailing gap penalties. For example, given two strings *ab* and *abcb*, we obtain the same global alignment score of $2m+2g$ for the alignments *ab- -* with *abcb*, versus *a- -b* with *abcb*. But for semi-global alignment, the similarity score is $2m$ for the first and $2m + 2g$ for the second, so we prefer the first since g takes a negative value.

The quality of sequence alignment depends critically on the particular parameters (weightings) we use. For our use, the major concern is deciding how many gaps to allow in order to gain a match. For example, when globally aligning *ab* with *bc*, two possible alignment results are *ab* with *bc* (score $n + n$) or *ab-* with *-bc* (score $g + m + g$). The three parameters will have a combined linear relationship (since we add combinations of them linearly to obtain a total score), so we proceed by fixing n and g (to 0 and -1, respectively), and adjusting m for different situations.

For global alignment—which we use to align two sequences before comparing them and locating length, cookie, and *don't-care* fields—we set $m = 1$ to avoid alignments like *ab-* with *-bc*. For semi-global alignment—used during replay to decide whether received data is complete—we set m to the length difference of the two sequences. The notion here is that if the last character of the received data matches the last one in the ADU from the script, then m is large enough to offset the gap penalty caused by aligning the characters together. However, if the received data is indeed incomplete, its better match to only the first part of the ADU will still win out. Using the semi-global alignment example above, we will set $m = 2$ (due to the length of *ab* vs. *abcb*), and hence still find the best alignment with *abcb* to be *ab- -* rather than *a- -b*.

Finally, we make a powerful refinement to Needleman-Wunsch for pinpointing endpoint-address and/or argument fields in a received ADU: we modify the algorithm to work with a *pairwise constraint matrix*. The matrix specifies whether the ith element of the first sequence can or cannot be aligned with the jth element of the second sequence. We dynamically generate this matrix based on the structure of the data from the primary dialog. For example, if the data includes an endpoint-address field represented as a dotted-quad IP address, then we add entries in the matrix prohibiting those digits from being matched with non-digits in the second data stream, and prohibiting the embedded "."s from being matched to anything other than "."s. This modification significantly improves the efficacy of the alignment algorithm in the presence of *don't-care* fields.

4.3.2 Removing overlap

When searching for dynamic fields in an ADU, sometimes we find fields that overlap with one another. We remove these using a greedy algorithm. For each overlapping dynamic field, we set the penalty for removing it as the number of bytes we would lose from the union set of all dynamic fields. (So, for example, a dynamic field that is fully embedded within another dynamic field has a penalty of 0.) We then select the overlapping field with the least penalty, remove it, and repeat the process until there is no overlap.

4.3.3 Handling large ADUs

ADUs can be very large, such as when transferring a large data item using a single Windows CIFS, NFS or FTP message. RolePlayer cannot ignore these ADUs when searching for dynamic fields because there may exist dynamic fields embedded within them—generally at the beginning. For example, an NFS "READ Reply" response comes with both the read status and the corresponding file data, and includes a cookie field containing a transaction identifier. However, the complexity of sequence alignment is $O(MN)$ for sequences of lengths M and N, making its application intractable for large sequences. RolePlayer avoids this problem by considering only fixed-size byte sequences at the beginning of large ADUs.

Protocol	Initiator Program	Responder Program	# Connections	# ADUs	# Initiator Fields		# Responder Fields	
					received	sent	received	sent
SMTP	manual	Sendmail	1	13	22	3	3	3
DNS	nslookup	BIND	1	2	8	0	0	1
HTTP	wget	Apache	1	2	10	0	0	0
TFTP	*W32.Blaster*	Windows XP	3	34	5	1	1	1
FTP	wget	ProFTPD	2	18	12	0	0	2
NFS	*mount*	Linux Fedora NFS	9	36	34	12	46	23
CIFS	*W32.Randex.D*	Windows XP	6	86	101	65	80	63

Table 1. Summary of evaluated applications and the number of dynamic fields in data received or sent by RolePlayer during either initiator or responder replay.

5 RolePlayer evaluation

We implemented RolePlayer in 4,300 lines of C code under Linux, using libpcap [11] to capture traffic and the standard socket API to generate traffic. Thus, RolePlayer only needs root access if it needs to send traffic from a privileged port. We tested the system on a variety of protocols widely used in malicious attacks and network applications. Table 1 summarizes our test suite. RolePlayer can successfully replay both the initiator and responder sides of all of these dialogs.

5.1 Test environment

We conducted our evaluation in an isolated testbed consisting of a set of nodes running on VMWare Workstation [23] interconnected using software based on Click [8]. We used VMWare Workstation's support for multiple guest operating systems and private networks between VM instances to construct different, contained test configurations, with the Click system redirecting malware scans to our chosen target systems. We gave each running VM instance a distinct IP address and hostname, and used non-persistent virtual disks to allow recovery from infection. For the tests we used both Windows XP Professional and Fedora Core 3 images, to verify that replay works for both Windows and Linux services. RolePlayer itself ran on the Linux host system rather than within a virtual machine, enabling it to communicate with any virtual machine on the system.

5.2 Simple protocols

Our simplest tests were for SMTP, DNS, and HTTP replay. For testing SMTP, we replayed an email dialog with RolePlayer changing the recipient's email address (an "argument" dynamic field). In one instance, RolePlayer itself made this change as the session initiator; in the other, it played the role of the responder (SMTP server).

For DNS, RolePlayer correctly located the transaction ID embedded within requests, updating it when replaying the responder (DNS server). The HTTP dialog was similarly simple, being limited to a single request and response. Since the request did not contain a cookie field, replaying trivially consisted of purely resending the same data as seen originally (though RolePlayer found some *don't-care* fields in the HTTP header of the response message).

5.3 Blaster (TFTP)

As discussed in the introduction, Blaster [24] attacks its victim using a DCOM RPC vulnerability. After attacking, it causes the victim to initiate a TFTP transfer back to the attacker to download an executable, which it then instructs the victim to run. A Blaster attack session does not contain any length fields or hostnames, so we can replay it without needing a secondary application dialog or hostname information.

RolePlayer can replay both sides of the dialog. As an initiator, we successfully infected a remote host with Blaster. As a fake victim, we tricked a live copy of Blaster into going through the full set of infection steps when it probed a new system.

When replaying the initiator, RolePlayer found five dynamic fields in received ADUs. Of these, it correctly deemed four as *don't-care* fields. These were each part of a confirmation message, specifying information such as data transfer size, time, and speed. The single dynamic field found and updated was the IP address of the initiator, necessary for correct operation of the injected TFTP command.

When replaying the responder, RolePlayer found only a single dynamic field, the worm's IP address, again necessary to correctly create the TFTP channel.

5.4 FTP

To test FTP replay, we used the wget utility as the client program to connect to two live FTP servers, *fedora.bu.edu* and *mirrors.xmission.com*. We collected two sample application dialogs using the command wget ftp://ftp-server-name/fedora/core/3/i386/os/Fedora/RPMS/crontabs-1.10-7.noarch.rpm. In both cases, wget used passive FTP, which uses dynamically created ports on the server side. When acting as the initiator, we replayed the *fedora.bu.edu* dialog over a live session to *mirrors.xmission.com*, and vice versa. There were no length fields, so we did not need a secondary dialog (nor hostnames). In both cases, RolePlayer successfully downloaded the file.

The system found 12 dynamic fields in the ADUs it received. Among them, the only two meaningful ones were endpoint-address fields: the server's IP address and the port of the FTP data-transfer channel. The rest arose due to differences in the greeting messages and authentication responses. RolePlayer recognized these as *don't-care*'s and did not update them.

When replaying the responder, wget successfully downloaded the file from a fake RolePlayer server pretending to be either *fedora.bu.edu* or *mirrors.xmission.com*, with the same two endpoint-address fields updated in ADUs sent by the replayer.

We tested support for argument fields by specifying the filename crontabs-1.10-7.noarch.rpm as an argument. When replaying the initiator, we replaced this with pyorbit-devel-2.0.1-1.i386.rpm, another file in the same directory. RolePlayer successfully downloaded the new file from *fedora.bu.edu* using the script from *mirrors.xmission.com*. Since the two files are completely different, the system found many *don't-care* fields. None of these affected the replay because they did not meet the conditions for updating cookie fields. We also confirmed that RolePlayer can replay non-passive FTP dialogs successfully.

5.5 NFS

We tested the NFS protocol running over SunRPC using two different NFS servers. We used the series of commands mount, ls, cp, umount to mount an NFS directory, copy a file from it to a local directory, and unmount it. We used one NFS server for collecting the primary application dialog and a second as the target for replaying the initiator. The NFS session consisted of nine TCP connections, including interactions with three daemons running on the NSF server: portmap, nfs, and mountd. The first two ran on 111/tcp and 2049/tcp, while mountd used a dynamic service port. As is usually the case with NFS access, in the session both of the latter two ports were found via RPCs to portmap.

When replaying the initiator, RolePlayer found 34 dynamic fields in received ADUs, and changed 12 fields in the ADUs it sent. When replaying the responder, RolePlayer found 46 dynamic fields in received ADUs, and changed 23 fields in the ADUs it sent. The cookie fields concerned RPC call IDs.

RolePlayer successfully replayed both the initiator side (receiving the directory listing and then the requested file) and the responder side (sending these to a live client, which correctly displayed the listing and copied the file).

5.6 Randex (CIFS/SMB)

To test RolePlayer's ability to handle a complex protocol while interacting with live malware, we used the *W32.Randex.D* worm [25]. This worm scans the network for SMB/CIFS shares with weak administrator passwords. To do so, it makes numerous SMB RPC calls (see Figure 5 in Appendix B). When it finds an open share, it uploads a malicious executable msmgri32.exe. In our experiments, we configured the targeted Windows VM to accept blank passwords, and turned off its firewall so it would accept traffic on ports 135/tcp, 139/tcp, 445/tcp, 137/udp, and 138/udp.

To collect sample application dialogs, we manually launched a malware executable from another Windows VM, redirecting its scans to the targeted Windows VM, recording the traffic using tcpdump. We captured two attacks because replaying CIFS requires a secondary application dialog to locate the numerous length fields.

There are six connection in *W32.Randex.D*'s attack, all started by the initiator. Of these, two are connections to 139/tcp which are reset by the initiator immediately after it receives a SYN-ACK from the responder. One connects to 80/tcp (HTTP), reset by the responder because the victim did not run an HTTP server. The remaining three connections were all to 445/tcp. The worm uses the first of these to detect a possible victim; it does not transmit any application data on this connection. The worm uses the second to enumerate the user account list on the responder via the SAMR named pipe. The final connection uploads the malicious executable to \Admin$\system32\msmsgri32.exe via the Admin named pipe. (More details are in Appendix B.)

When replaying the initiator, RolePlayer found 101 dynamic fields in received ADUs, and changed 65 fields in the ADUs it sent. When replaying the responder, it found 80 dynamic fields in received ADUs, and changed 63 fields in the ADUs it sent. (The difference in the number of fields is because some dynamic fields remain the same when they come from the replayer rather than the worm. For example, the responder chooses the context handle of the SAMR named pipe; when replaying the responder, the replayer just uses the same context handle as in the primary application dialog.) Considering ADUs in both directions, there were 21 endpoint-address fields, 76 length fields, and 32 cookie fields. The cookie fields reflect such information as the context handles in SAMR named pipes and the client process IDs.

As with our Blaster experiment, RolePlayer successfully infected a live Windows system with *W32.Randex.D* when replaying the initiator side, and, when replaying the responder, successfully drove a live, attacking instance of the worm through its full set of infection steps.

5.7 Discussion

From the experiments we can see that it is necessary to locate and update all dynamic fields—endpoint-address, cookie, and length fields— for replaying protocols successfully, while argument fields are important for extending RolePlayer's functionality. TFTP, FTP, and NFS require correct manipulation of endpoint-address fields. DNS, NFS, and CIFS also rely on correct identification of cookie files. CIFS has numerous length fields within a single application dialog. Leveraging argument fields, RolePlayer can work as a low-cost client for SMTP, FTP, or DNS.

While RolePlayer can replay a wide class of application protocols, its coverage is not universal. In particular, it cannot accommodate protocols with time-dependent state, nor those that use cryptographic authentication or encrypted traffic, although we can envision dealing with the latter by introducing application-specific extensions to provide RolePlayer with access to a session's clear-text dialog. Another restriction is that the live peer with which RolePlayer engages must behave in a fashion consistent with the "script" used to configure RolePlayer. This requirement is more restrictive than simply that the live peer follows the application protocol: it must also follow the particular path present in the script.

Since RolePlayer keeps some dynamic fields unchanged as in the primary dialog, it is possible for an adversary to detect the existence of a running RolePlayer by checking if certain dynamic fields are changed among different sessions. For example, RolePlayer will always open the same port for the data channel when replaying the responder of an FTP dialog, and it will use the same context handles in SAMR named pipes when replaying the responder of a CIFS dialog. Another possible way to detect RolePlayer is to discover inconsistencies between the operating system the application should be running on versus the operating system RolePlayer is running on, by fingerprinting packet headers [26]. In the future, we plan to address these problems by randomizing certain dynamic fields and by manipulating packet headers to match the expected operating system.

6 Summary

We have presented RolePlayer, a system that, given examples of an application session, can mimic both the initiator and responder sides of the session for a wide variety of application protocols. We can potentially use such replay for recognizing malware variants, determining the range of system versions vulnerable to a given attack, testing defense mechanisms, and filtering multi-step attacks. However, while for some application protocols replay can be essentially trivial—just resend the same bytes as recorded for previously seen examples of the session—for other protocols replay can require correctly altering numerous fields embedded within the examples, such as IP addresses, hostnames, port numbers, transaction identifiers and other opaque cookies, as well as length fields that change as these values change.

We might therefore conclude that replay inevitably requires building into the replayer specific knowledge of the applications it can mimic. However, one of the key properties of RolePlayer is that it operates in an *application-independent* fashion: the system does not require any specifics about the particular application it mimics. It instead uses extensions of byte-stream alignment algorithms from bioinformatics to compare different instances of a session to determine which fields it must change to successfully replay one side of the session. To do so, it needs only two examples of the particular session; in some cases, a single example suffices.

RolePlayer's understanding of network protocols is very limited—just knowledge of a few low-level syntactic conventions, such as common representations of IP addresses and the use of length fields to specify the size of subsequent variable-length fields. (The only other information RolePlayer requires—depending on the application protocol—is context for the example sessions, such as the domain names of the participating hosts and specific arguments in requests or responses if we wish to change these when replaying.) This information suffices for RolePlayer to heuristically detect and adjust network addresses, ports, cookies, and length fields embedded within the session, including for sessions that span multiple, concurrent connections.

We have successfully used RolePlayer to replay both the initiator and responder sides for a variety of network applications, including: SMTP, DNS, HTTP; NFS, FTP and CIFS/SMB file transfers; and the multi-stage infection processes of the Blaster and W32.Randex.D worms. The latter require correctly engaging in connections that, within a single session, encompass multiple protocols and both client and server roles. An important item of future work is to identify and test additional, complex application protocols, to gain a deeper understanding of the generality of our approach.

Our next step is to deploy RolePlayer in a large honeypot installation, for purposes of both filtering out known attacks (by replaying enough server-side responses to the attackers to check whether their behavior matches a known attack session) and replaying successful attacks against other possible victim configurations. This latter has the potential to enable us to automatically determine the full range of systems vulnerable to previously unseen "zero day" exploits as soon as they exploit our honeypot system.

Acknowledgments

We would like to thank Martin Casado, Christian Kreibich, Sridhar Machiraju, and Scott Shenker for their helpful comments on a draft of this paper. We would also like to thank Ruoming Pang and Vinod Yegneswaran for their help in early discussions and providing network traces, Carey Nachenberg and Vincent Weafer for technical assistance, and the anonymous reviewers for their insightful comments. This work was supported by the National Science Foundation under grants NSF-0433702 and STI-0334088, for which we are grateful.

References

[1] D. F. Bacon and S. C. Goldstein. Hardware-assisted replay of multiprocessor programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1991.

[2] M. Beddoe. The protocol informatics project. http://www.baselineresearch.net/PI/.

[3] Y.-C. Cheng, U. Holzle, N. Cardwell, S. Savage, and G. Voelker. Monkey see, monkey do: A tool for tcp tracing and replaying. In *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004.

[4] Cybertrace. http://www.cybertrace.com/ctids.html.

[5] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, December 2002.

[6] Honeynet. The honeynet project. http://www.honeynet.org/.

[7] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX Annual Technical Conference*, April 2005.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[9] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. pages 471–482, April 1987.

[10] D. Libes. expect: Curing those uncontrollable fits of interaction. In *Proceedings of the Summer 1990 USENIX Conference*, pages 183–192, June 1990.

[11] libpcap. http://www.tcpdump.org/.

[12] McAfee Inc. McAfee Security Forensics. http://networkassociates.com/us/products/mcafee/forensics /security_forensics.htm.

[13] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. 48:443–453, 1970.

[14] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, February 2005.

[15] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of Internet background radiation. In *Proceedings of Internet Measurement Conference*, October 2004.

[16] N. Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.

[17] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the 1996 Conference on Programming Language Design and Implementation*, pages 258–266, May 1996.

[18] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A light-weight rollback and deterministic replay extension for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference*, June 2004.

[19] Synmantec. Decoy server product sheet. http://www.symantec.com/.

[20] Tcpreplay: Pcap editing and replay tools for *NIX. http://tcpreplay.sourceforge.net.

[21] A. Turner. Flowreplay design notes. http://www.synfin.net/papers/flowreplay.pdf.

[22] Unicode. http://www.unicode.org/.

[23] VMWare Inc. http://www.vmware.com/.

[24] W32.Blaster.Worm. http://securityresponse.symantec.com /avcenter/venc/data/w32.blaster.worm.html.

[25] W32.Randex.D. http://securityresponse.symantec.com /avcenter/venc/data/w32.randex.d.html.

[26] M. Zalewski. P0f: A passive OS fingerprinting tool. http://lcamtuf.coredump.cx/p0f.shtml.

A The Toy Service Port Discovery Protocol

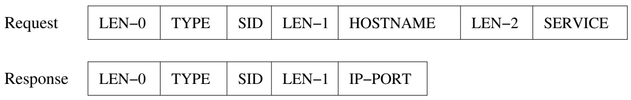


Figure 4. Message format for the toy Service Port Discovery protocol.

We define the toy Service Port Discovery (SPD) protocol for illustrating the algorithms in RolePlayer. In SPD, a client sends a request message to a server to ask for the port number of a service. The server's response contains the IP address and the port number of the requested service.

These two messages have the formats shown in Figure 4. Requests have 7 fields: LEN-0 (1 byte) holds the length of the message. TYPE (1 byte) indicates the message type, with a value of 1 indicating a request and 2 a response message. SID (2 bytes) is session/transaction identifier, which the server must echo in its response. LEN-1 (1 byte) stores the length of the client hostname, HOSTNAME, which the server logs. LEN-2 (1 byte) stores the length of the service name, SERVICE.

Responses have 5 fields. LEN-0, TYPE and SID have the same meanings as in requests. LEN-1 (1 byte) stores the length of the IP-PORT field, which holds the IP address and port number of the requested service, in a comma-separated format. For example, 1.2.3.4:567 is expressed as "`0x31 0x2c 0x32 0x2c 0x33 0x2c 0x34 0x2c 0x32 0x2c 0x35 0x35`".

B W32.Randex.D

We show the application-level conversations of the W32.Randex.D worm on port 445/tcp in Figure 5 (reproduced with permission from [15]).

To demonstrate the function of RolePlayer, we select six consecutive messages from the conversation of W32.Randex.D (shown in bold-italic in Figure 5), consisting of SAMR Connect4 request/response, SAMR EnumDomains request/response, and SAMR LookupDomain request/response. We show the content of these messages from the primary, secondary, initiator-based replay, and responder-based replay dialogs in Figure 6. To fit each message more compactly, we present them in the following format:

1. We split each message based on dynamic fields.

2. XY means we skipped Y bytes from protocol X (N for NetBIOS, S for SMB, R for DCE-RPC, M for Security Account Manager), since they do not change between different dialogs. These represent the fixed fields as part of the dialog.

3. We present endpoint-address fields such as IP addresses and hostnames in ASCII format in bold-italic. Three hostnames appear in the messages: "*hone*", "*host02*", and "*hostpeer*".

4. We show length fields in decimal, with a gray background. For example, "180" and "96" in the first message of the primary dialog are length fields.

5. We show cookie fields and *don't-care* fields, including the client process IDs and the SAMR context handles, in octets. For example, "0388" in the first message of the primary dialog is a client process ID. For convenience, we highlight in bold the cookies in the primary and secondary dialog which require dynamic updates during the replay process.

Note that RolePlayer located two cookie fields from the SAMR context handles because part of the handles does not change between dialogs (e.g., the middle portion was constant in all the dialogs).

```
-> SMB Negotiate Protocol Request
<- SMB Negotiate Protocol Response
-> SMB Session Setup AndX Request
<- SMB Session Setup AndX Response
-> SMB Tree Connect AndX Request,
        Path:  \\XX.128.18.16\IPC$
<- SMB Tree Connect AndX Response
-> SMB NT Create AndX Request, Path: \samr
<- SMB NT Create AndX Response
-> DCERPC Bind: call_id: 1 UUID: SAMR
<- DCERPC Bind_ack:
-> SAMR Connect4 request
<- SAMR Connect4 reply
-> SAMR EnumDomains request
<- SAMR EnumDomains reply
-> SAMR LookupDomain request
<- SAMR LookupDomain reply
-> SAMR OpenDomain request
<- SAMR OpenDomain reply
-> SAMR EnumDomainUsers request
```

Now start another session, connect to the SRVSVC pipe and issue NetRemoteTOD (get remote Time of Day) request

```
-> SMB Negotiate Protocol Request
<- SMB Negotiate Protocol Response
-> SMB Session Setup AndX Request
<- SMB Session Setup AndX Response
-> SMB Tree Connect AndX Request,
        Path: \ \XX.128.18.16\IPC$
<- SMB Tree Connect AndX Response
-> SMB NT Create AndX Request, Path: \srvsvc
<- SMB NT Create AndX Response
-> DCERPC Bind: call_id: 1 UUID: SRVSVC
<- DCERPC Bind_ack: call_id: 1
-> SRVSVC NetrRemoteTOD request
<- SRVSVC NetrRemoteTOD reply
-> SMB Close request
<- SMB Close Response
```

Now connect to the ADMIN share and write the file

```
-> SMB Tree Connect AndX Request, Path: \\XX.128.18.16\ADMIN$
<- SMB Tree Connect AndX Response
-> SMB NT Create AndX Request,
    Path:\system32\msmsgri32.exe  <<<===

<- SMB NT Create AndX Response, FID: 0x74ca
-> SMB Transaction2 Request SET_FILE_INFORMATION
<- SMB Transaction2 Response SET_FILE_INFORMATION
-> SMB Transaction2 Request QUERY_FS_INFORMATION
<- SMB Transaction2 Response QUERY_FS_INFORMATION
-> SMB Write Request
....
```

Figure 5. The application-level conversation of W32.Randex.D.

```
A->V   N1|180|S26|0388|S7|96|S20|96|S8|113|S16|R8|96|R6|72|R4|0014CCB8|18|M4|18|M4
            |144.165.114.119|M20
A<-V   N4|S26|0388|S28|R24|M16|0000000092F3E82470FDD91195F8000C295763F7|M4
A->V   N4|S26|0388|S56|R24|0000000092F3E82470FDD91195F8000C295763F7|M8
A<-V   N1|180|S26|0388|S7|124|S8|124|S6|125|S1|R8|124|DECRPC-6|100|R4|M4|000B0BB0
            |M4|000B27A0|M8|8|10|000B8D18|M8|000BC610|5|M4|4|hone|M36
A->V   N1|156|S26|0388|S7|72|S20|72|S8|89|S16|R8|72|R6|48|R4|M4
            |0000000092F3E82470FDD91195F8000C295763F7|8|10|001503F8|5|M4|4|hone
A<-V   N4|S26|0388|S28|R24|000B27A0|M32
```

(a) Primary Dialog

```
A->V   N1|172|S26|0474|S7|88|S20|88|S8|105|S16|R8|88|R6|64|R4|0014CCB8|14|M4|14|M4
            |48.196.8.48|M20
A<-V   N4|S26|0474|S28|R24|M16|000000006093917586FDD91195F8000C294A478F|M4
A->V   N4|S26|0474|S56|R24|000000006093917586FDD91195F8000C294A478F|M8
A<-V   N1|184|S26|0474|S7|128|S8|128|S6|129|S1|R8|128|DECRPC-6|104|R4|M4|000B0BB0
            |M4|000B6380|M8|12|14|000B76C0|M8|000C9FA8|7|M4|6|host02|M36
A->V   N1|160|S26|0474|S7|76|S20|76|S8|89|S16|R8|76|R6|52|R4|M4
            |000000006093917586FDD91195F8000C294A478F|12|14|001503F8|7|M4|6|host02
A<-V   N4|S26|0474|S28|R24|000B27A0|M32
```

(b) Secondary Dialog

```
A->V   N1|176|S26|0388|S7|92|S20|92|S8|109|S16|R8|92|R6|68|R4|0014CCB8|16|M4|16|M4
            |192.168.170.3|M20
A<-V   N4|S26|0388|S28|R24|M16|0000000018B30AD10BFDD91195F8000C293573E4|M4
A->V   N4|S26|0388|S56|R24|0000000018B30AD10BFDD91195F8000C293573E4|M8
A<-V   N1|188|S26|0388|S7|132|S8|132|S6|133|S1|R8|132|DECRPC-6|108|R4|M4|000B0BB0
            |M4|000B9358|M8|16|18|000BEF40|M8|000B6BA0|9|M4|8|hostpeer|M36
A->V   N1|164|S26|0388|S7|80|S20|80|S8|89|S16|R8|80|R6|56|R4|M4
            |0000000018B30AD10BFDD91195F8000C293573E4|16|18|001503F8|9|M4|8|hostpeer
A<-V   N4|S26|0388|S28|R24|000BEF40|M32
```

(c) Initiator-based Replay Dialog

```
A->V   N1|176|S26|0608|S7|92|S20|92|S8|109|S16|R8|92|R6|68|R4|0014CCB8|16|M4|16|M4
            |169.91.250.93|M20
A<-V   N4|S26|0608|S28|R24|M16|0000000092F3E82470FDD91195F8000C295763F7|M4
A->V   N4|S26|0608|S56|R24|0000000092F3E82470FDD91195F8000C295763F7|M8
A<-V   N1|180|S26|0608|S7|124|S8|124|S6|125|S1|R8|124|DECRPC-6|100|R4|M4|000B0BB0
            |M4|000B27A0|M8|8|10|000B8D18|M8|000BC610|5|M4|4|hone|M36
A->V   N1|156|S26|0608|S7|72|S20|72|S8|89|S16|R8|72|R6|48|R4|M4
            |0000000092F3E82470FDD91195F8000C295763F7|8|10|00150A88|5|M4|4|hone
A<-V   N4|S26|0608|S28|R24|000B27A0|M32
```

(d) Responder-based Replay Dialog

Figure 6. A portion of the application-level conversation of W32.Randex.D. Endpoint-address fields are in bold-italic, cookie fields are in bold, and length fields are in gray background. In the initiator-based and responder-based replay dialogs, updated fields are in black background.