US 20090083854A1

(19) United States

(12) Patent Application Publication

(10) Pub. No.: US 2009/0083854 A1

(43) Pub. Date: Mar. 26, 2009

Bozanich et al.

(54) **SYNTAX-BASED SECURITY ANALYSIS USING DYNAMICALLY GENERATED TEST CASES**

(75) Inventors: **Adam Bozanich**, San Francisco, CA (US); **Kowsik Guruswamy**, Sunnyvale, CA (US); **Marshall A. Beddoe**, San Francisco, CA (US)

Correspondence Address:
FENWICK & WEST LLP
SILICON VALLEY CENTER, 801 CALIFORNIA STREET
MOUNTAIN VIEW, CA 94041 (US)

(73) Assignee: **MU SECURITY, INC.**, Sunnyvale, CA (US)

(21) Appl. No.: **11/858,779**

(22) Filed: **Sep. 20, 2007**

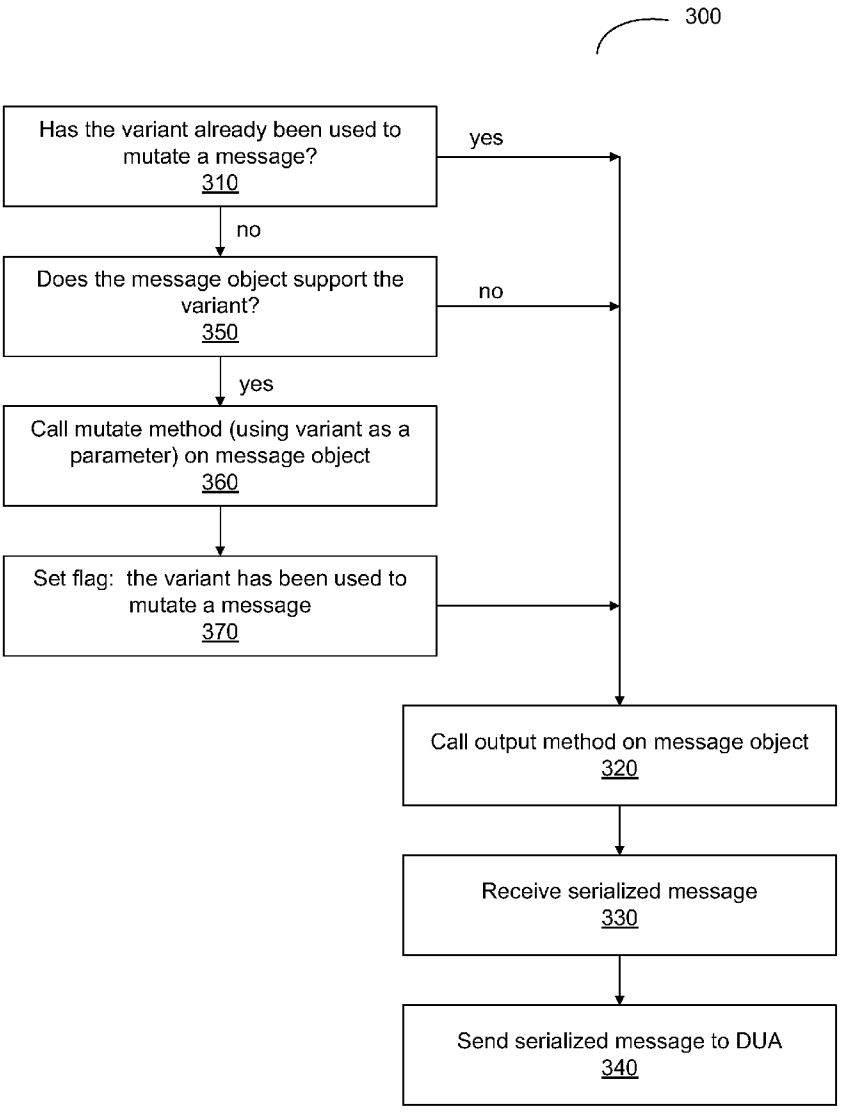
Publication Classification

(51) **Int. Cl.**
G06F 21/00 (2006.01)

(52) **U.S. Cl.** **726/23**

(57) **ABSTRACT**

A security analysis methodology is used to analyze the security of a device-under-analysis (DUA) with respect to a particular protocol message exchange. First, the mutation points that exist in the message exchange are determined. Then, the message exchange is executed multiple times—once for each mutation point. Each execution applies the mutation associated with that particular mutation point (e.g., a particular message during the exchange is modified in a particular way) to create a mutated message exchange. In other words, each message exchange with an applied mutation point corresponds to a test case.



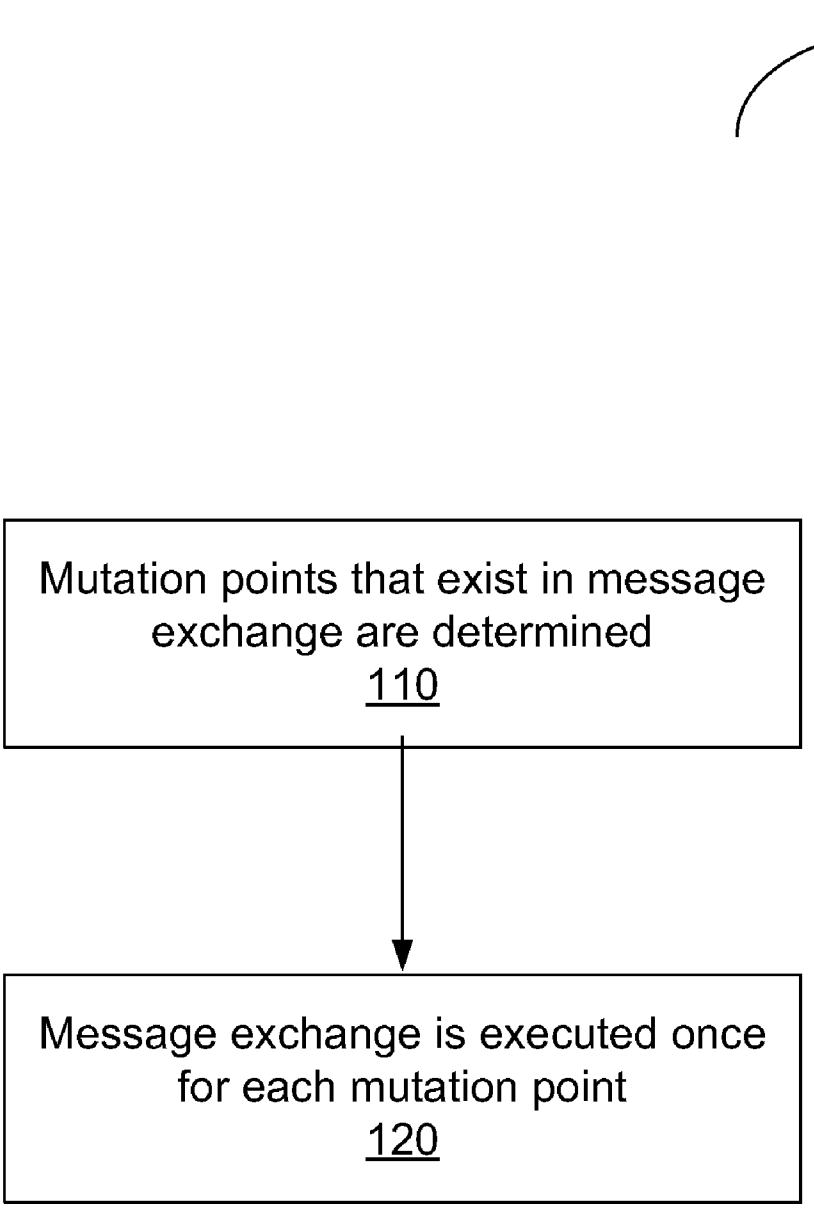

100
Mutation points that exist in message exchange are determined
110
Message exchange is executed once for each mutation point
120

FIG. 1

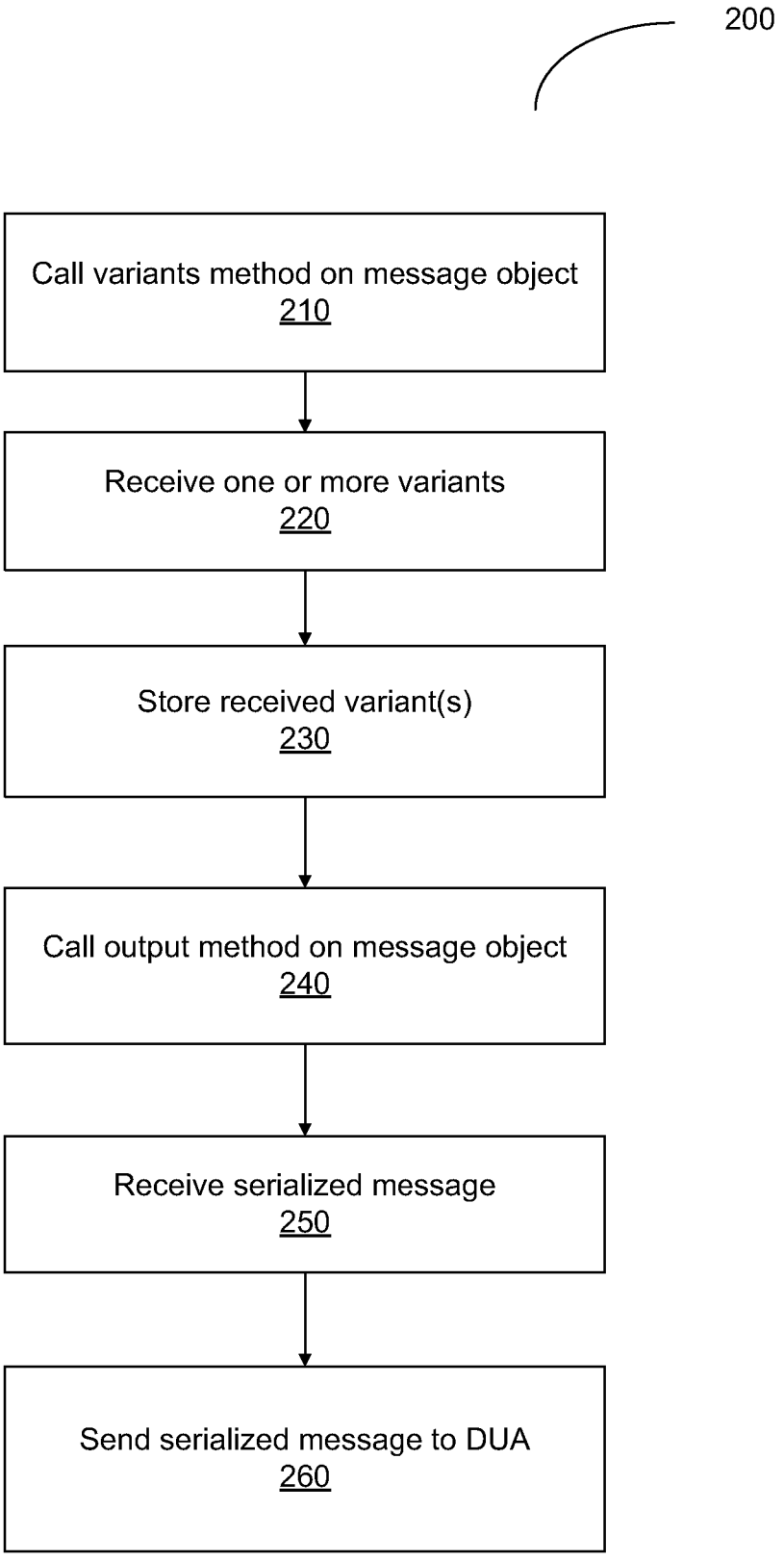

200
Call variants method on message object
210
Receive one or more variants
220
Store received variant(s)
230
Call output method on message object
240
Receive serialized message
250
Send serialized message to DUA
260

FIG. 2

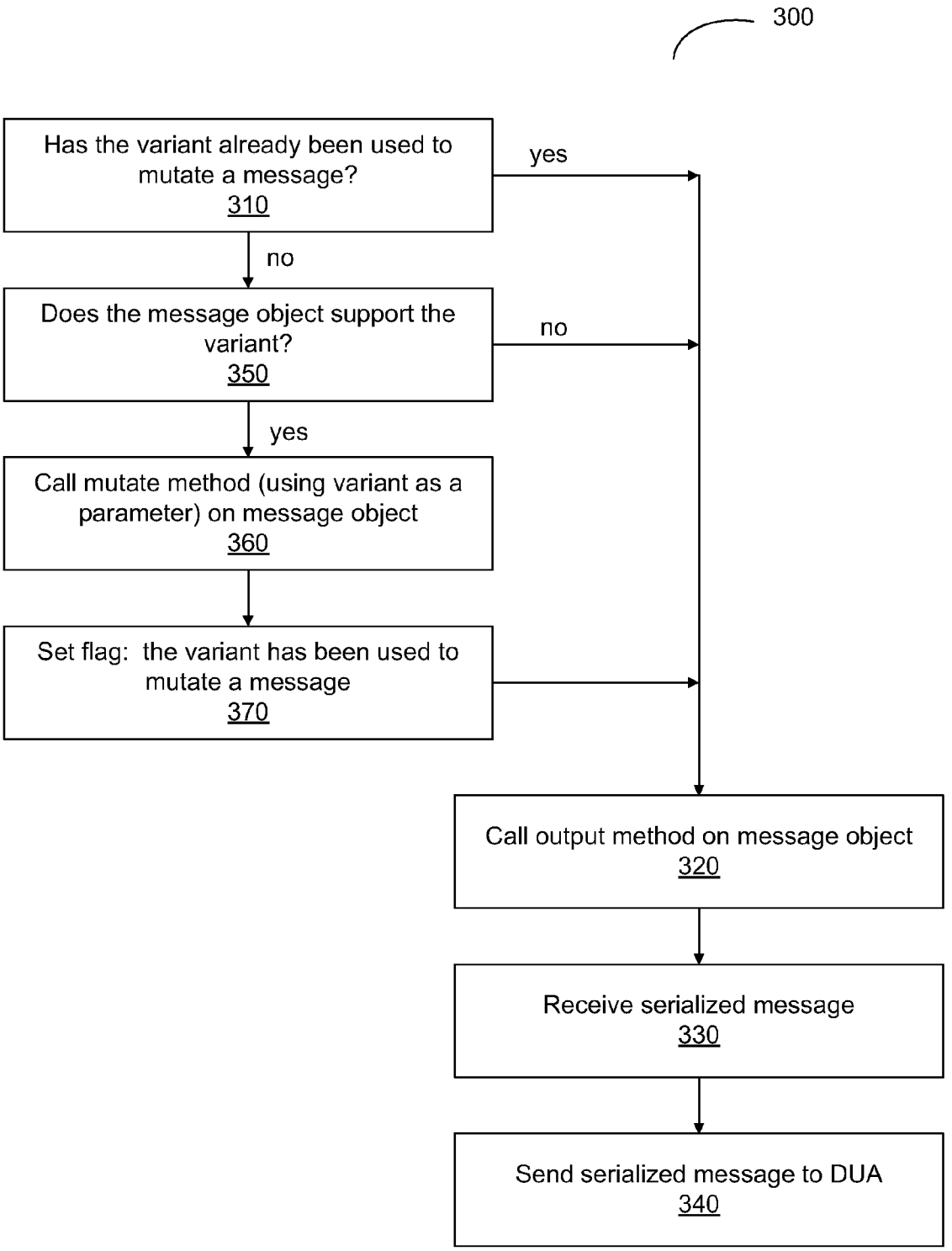

300
Has the variant already been used to mutate a message?
310
yes
no
Does the message object support the variant?
350
no
yes
Call mutate method (using variant as a parameter) on message object
360
Set flag: the variant has been used to mutate a message
370
Call output method on message object
320
Receive serialized message
330
Send serialized message to DUA
340

FIG. 3

SYNTAX-BASED SECURITY ANALYSIS USING DYNAMICALLY GENERATED TEST CASES

REFERENCE TO RELATED APPLICATIONS

[0001] This application is related to the following utility applications, which are hereby incorporated by reference in their entirety: U.S. application Ser. No. 11/351,403, filed on Feb. 10, 2006, entitled "Platform for Analyzing the Security of Communication Protocols and Channels"; U.S. application Ser. No. 11/514,809, filed on Sep. 1, 2006, entitled "Automated Generation of Attacks for Analyzing the Security of Communication Protocols and Channels"; and U.S. application Ser. No. 11/745,338, filed on May 7, 2007, entitled "Modification of Messages for Analyzing the Security of Communication Protocols and Channels."

BACKGROUND

[0002] The present invention relates to security analysis and/or syntax testing of hardware and software and automated generation and execution of test cases.

[0003] Computerized communication, whether it occurs at the application level or at the network level, generally involves the exchange of data or messages in a known, structured format (a "protocol"). Specifically, a protocol refers to what is being communicated (for example, the data or message content). Types of protocols include, for example, networking protocols (including network packets), application program interfaces (APIs; including API calls, remote method invocation (RMI), and remote procedure call (RPC)), and file formats.

[0004] Software applications and hardware devices that rely on these formats can be vulnerable to various attacks that are generally known as "protocol abuse." Protocol abuse consists of sending messages that are invalid or malformed with respect to a particular protocol ("protocol anomalies") or sending messages that are well-formed but inappropriate based on a system's state. Messages whose purpose is to attack a system are commonly known as malicious network traffic.

[0005] Various systems have been developed that identify or detect attacks when they occur. This functionality, which is known as intrusion detection, can be implemented by a system that is either passive or active. A passive intrusion detection system (IDS) will merely detect an attack, while an active IDS will attempt to thwart the attack. Note that an IDS reacts to an actual attack. While an IDS might be able to detect an attack, it does not change the fact that an attack has occurred and might have damaged the underlying system.

[0006] A proactive solution to the attack problem is to analyze a system ahead of time to discover or identify any vulnerabilities. This way, the vulnerabilities can be addressed before the system is deployed or released to customers. This process, which is known as "security analysis," can be performed using various methodologies. One methodology for analyzing the security of a device-under-analysis (DUA) is to treat the DUA as a black box. Under this methodology, the DUA is analyzed via the interfaces that it presents to the outside world. As a result, it is not necessary to access the source code or object code comprising the DUA.

[0007] For example, a security analyzer executes a test case by sending one or more messages (test messages) to the DUA. The analyzer then observes the DUA's response. A response can include, for example, registering an error or generating a message (response message). The DUA can then send the response message to the analyzer. Depending on the analysis test case being performed, the analyzer might send another test message to the DUA upon receiving the response message from the DUA. The test messages and response messages can be analyzed to determine whether the DUA operated correctly.

[0008] In one embodiment, a test message is "improper" in that its structure does not conform to the appropriate protocol. Protocol structure (also known as syntax) refers to the layout of a message, such as its fields, arguments, or parameters, and its possible length. In this embodiment, security analysis of the DUA would be similar to syntax-based vulnerability testing of the DUA's implementation of the protocol.

[0009] Each test case that is executed helps to analyze a different aspect of the DUA's security. Thus, in order to analyze the security of a DUA, it is necessary to execute several different test cases. While test cases can be created manually, it is more efficient to create them in an automated fashion (see, for example, U.S. application Ser. No. 11/514,809, filed on Sep. 1, 2006, entitled "Automated Generation of Attacks for Analyzing the Security of Communication Protocols and Channels" and U.S. application Ser. No. 11/745,338, filed on May 7, 2007, entitled "Modification of Messages for Analyzing the Security of Communication Protocols and Channels").

[0010] In the past, a complete test case was created before the test case was used for testing. For example, consider a test case that calls for an exchange of messages: a first message (test message) sent to the DUA, a second message (response message) received from the DUA, and a third message (test message) sent to the DUA. In the past, the test case would define the first and third messages and cause them to be sent at the appropriate times. Since the third message was defined when the test case was created, it remained the same every time the test case was executed, regardless of the contents of the second message.

[0011] But what if the protocol called for the contents of the third message to depend on the contents of the second message? In this situation, the test case would have to parse the second message and then use this information to generate the third message. This would be difficult to achieve due to the dynamic nature of protocols, the ambiguity of protocol specifications, and the diversity of protocol implementations. As a result, static test cases can be insufficient to effectively perform syntax-based testing.

SUMMARY

[0012] Static test cases can be insufficient to effectively perform syntax-based testing. With a dynamic security analyzer, the generation of a test case is intertwined with the execution of that test case, and the generation can depend on the execution. Specifically, when a dynamic security analyzer begins executing a test case, the test case may or may not have been fully generated. This way, later messages in the test case message exchange can be generated based on earlier messages in the exchange. This enables the test case to take into account the dynamic nature of network protocols.

[0013] A security analysis methodology is used to analyze the security of a device-under-analysis (DUA) with respect to a particular protocol message exchange. First, the mutation points that exist in the message exchange are determined. Then, the message exchange is executed multiple times—

once for each mutation point. Each execution applies the mutation associated with that particular mutation point (e.g., a particular message during the exchange is modified in a particular way) to create a mutated message exchange. In other words, each message exchange with an applied mutation point corresponds to a test case.

[0014] In one embodiment, the first step of the methodology is achieved as follows: The complete message exchange is performed using proper (i.e., not mutated) messages. For example, a security analyzer exchanges messages with a DUA according to a particular protocol. During the performance of the exchange, the messages that are sent back and forth are observed, along with the fields that they contain. Each message is stored, including the values of its fields. Also, the mutation points of each message are stored. As mentioned above, some mutations target a message as a whole, while others target a field of a message. For a mutation that targets an entire message, it is sufficient to store an indication of the mutation. For a mutation that targets a field, an indication of the field is stored along with an indication of the mutation.

[0015] In one embodiment, the second step of the methodology is achieved as follows: During the performance of an exchange, the messages that are generated by the security analyzer are observed, along with the fields that they contain, before they are sent to the DUA. When a message or a field is observed that is due to be mutated (based on the particular mutation point currently being tested), that message or field is mutated before the message is sent to the DUA.

[0016] Other aspects of the invention include software, systems, components, and methods corresponding to the above, and applications of the above for purposes other than security analysis.

BRIEF DESCRIPTION OF THE DRAWINGS

[0017] The invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings in which like reference numerals refer to similar elements.

[0018] FIG. **1** illustrates a flowchart for a security analysis methodology, according to one embodiment of the invention.

[0019] FIG. **2** illustrates a flowchart for a portion of the enumeration phase that concerns sending a message to the DUA, according to one embodiment of the invention.

[0020] FIG. **3** illustrates a flowchart for a portion of the analysis phase that concerns sending a message to the DUA, according to one embodiment of the invention.

DETAILED DESCRIPTION

[0021] In the following description, "device," "device-under-analysis," and "DUA" represent software and/or hardware. Software includes, for example, applications, operating systems, and/or communications systems. Hardware includes, for example, one or more devices. A device can be, for example, a switch, bridge, router (including wireline or wireless), packet filter, firewall (including stateful or deep inspection), Virtual Private Network (VPN) concentrator, Network Address Translation (NAT)-enabled device, proxy (including asymmetric), intrusion detection/prevention system, or network protocol analyzer. A DUA can also be multiple devices that are communicatively coupled to form a system or network of devices. For example, a DUA can be two firewall devices that establish an encrypted tunnel between themselves.

[0022] A platform for analyzing the security of a DUA, called a "security analyzer," is described in U.S. application Ser. No. 11/351,403, filed on Feb. 10, 2006, entitled "Platform for Analyzing the Security of Communication Protocols and Channels" ("Previous Application"), which is hereby incorporated by reference in its entirety. In one embodiment, the security analyzer in the Previous Application (hereinafter referred to as a "static security analyzer") is an appliance that is capable of executing attacks (test cases) to analyze the security of the DUA. A static security analyzer executes a test case by sending one or more messages (test messages) to the DUA. In one embodiment, a message is a network packet or a protocol data unit (PDU). The static analyzer then observes the DUA's response. A response can include, for example, registering an error or generating a message (response message). The DUA can then send the response message to the static analyzer. Depending on the analysis test case being performed, the static analyzer might send another test message to the DUA upon receiving the response message from the DUA. The test messages and response messages can be analyzed to determine whether the DUA operated correctly.

[0023] Consider a sequence of messages that are exchanged between a first device and a second device. The messages that are exchanged are supposed to adhere to a particular protocol. Each message in the exchange could be mutated (modified) in various ways so that its structure and/or content does not conform to the appropriate protocol. The way in which the mutated message differs from a proper message is referred to as a "mutation." Each mutation can be thought of as a function or algorithm that is applied to a valid protocol message in order to generate an invalid protocol message.

[0024] Many categories of mutations exist. One category targets a message as a whole. For example, an extra field can be added to the message (e.g., by appending a random number or string) or a field can be removed from the message. Another category targets a field of the message. Mutations in the second category are usually specialized based on the data type of the value of that field. For example, if a field is supposed to indicate an Internet Protocol (IP) address, it can be attacked by setting its value to 0.0.0.0.

[0025] Appendix A contains exemplary data types, each of which can be associated with one or more mutations. Appendix B contains exemplary semantic rules, each of which specifies how to (correctly) generate a semantic element of a message (e.g., a value of a field) and can be susceptible to attack. Appendix C contains exemplary input/output (I/O) rules, each of which specifies a lower-layer protocol underlying a message (e.g., a transport-layer protocol) and can be susceptible to attack. Appendix D contains exemplary mutations.

[0026] Each point in a message exchange where a message can be mutated in a particular way is called a "mutation point." Since a single message can be mutated in many different ways and a message exchange consists of many different messages (and, thus, many different mutation points), the total number of test cases (i.e., possible message exchanges that use mutated messages) grows exponentially. Yet, each of these test cases should be generated and executed in order to analyze the security of a DUA. When the dynamic nature of

the DUA is taken into account (e.g., the types and numbers of messages that it can generate), the number of test cases becomes infinite.

[0027] In one embodiment, a new type of security analyzer (hereinafter referred to as a "dynamic security analyzer") is used that can generate a test case dynamically so that the test case does not need to be scripted prior to its execution. Recall that when a static security analyzer begins executing a test case (e.g., in order to perform syntax-based security analysis), the test case has already been fully scripted. Thus, the generation of a test case and the execution of that test case are completely independent of each other. Specifically, the way in which a test case is generated (e.g., manually or in an automated fashion and, if automated, how the automation works) is irrelevant to how the test case is executed and vice versa.

[0028] As described above, static test cases can be insufficient to effectively perform syntax-based testing. With a dynamic security analyzer, the generation of a test case is intertwined with the execution of that test case, and the generation can depend on the execution. Specifically, when a dynamic security analyzer begins executing a test case, the test case may or may not have been fully generated. This way, later messages in the test case message exchange can be generated based on earlier messages in the exchange. This enables the test case to take into account the dynamic nature of network protocols.

[0029] For example, consider a test case that calls for an exchange of messages: a first message (test message) sent to the DUA, a second message (response message) received from the DUA, and a third message (test message) sent to the DUA wherein the protocol calls for the contents of the third message to depend on the contents of the second message. When a dynamic security analyzer begins executing this test case, the third message has not yet been generated. Instead, the third message will be generated after the second message has been received (i.e., during execution of the test case).

I. Security Analysis Methodology

[0030] FIG. **1** illustrates a flowchart for a security analysis methodology, according to one embodiment of the invention. The security analysis methodology **100** is used to analyze a DUA's security with respect to a particular protocol message exchange. First, the mutation points that exist in the message exchange are determined **110**. Then, the message exchange is executed **120** multiple times—once for each mutation point. Each execution applies the mutation associated with that particular mutation point (e.g., a particular message during the exchange is modified in a particular way) to create a mutated message exchange. In other words, each message exchange with an applied mutation point corresponds to a test case.

[0031] I.A Determination of Mutation Points

[0032] The first step in security analysis methodology **100** is to determine **110** the mutation points that exist in a particular message exchange. In one embodiment, this is achieved as follows: The complete message exchange is performed using proper (i.e., not mutated) messages. For example, a security analyzer exchanges messages with a DUA according to a particular protocol.

[0033] During the performance of the exchange, the messages that are sent back and forth are observed, along with the fields that they contain. Each message is stored, including the values of its fields. Also, the mutation points of each message are stored. As mentioned above, some mutations target a message as a whole, while others target a field of a message. For a mutation that targets an entire message, it is sufficient to store an indication of the mutation. For a mutation that targets a field, an indication of the field is stored along with an indication of the mutation.

[0034] For example, if a message includes an IP address field, an indication is stored that this field can be mutated using IP address attacks (such as setting the address to 0.0.0.0, as mentioned above). As another example, if a message includes a length field, an indication is stored that this field can be mutated using length-related attacks (such as setting the length to a negative number).

[0035] Since the message exchange is occurring with the actual DUA that will be analyzed, the list of mutation points will be tailored specifically to that DUA and the testing environment. This phase is referred to as "enumeration" because the various mutation points are enumerated.

[0036] I.B Execution of Mutated Message Exchange

[0037] The second step in security analysis methodology **100** is to execute **120**, for each mutation point that was enumerated in the previous step, a mutated message exchange. This will ensure that all of the mutation points are tested. This phase is referred to as the analysis phase, since the DUA is being analyzed.

[0038] During the performance of an exchange, the messages that are generated by the security analyzer are observed, along with the fields that they contain, before they are sent to the DUA. When a message or a field is observed that is due to be mutated (based on the particular mutation point currently being tested), that message or field is mutated before the message is sent to the DUA.

[0039] Since the DUA is deterministic, a message exchange should proceed in exactly the same way that it did during the enumeration phase. Thus, the list of mutation points that was generated during the enumeration phase should also apply to later message exchanges during the analysis phase. Also, mutations can be based on messages that were stored during the enumeration phase. This means that messages can be customized specifically to this DUA and its responses.

[0040] Since the original (non-mutated) message exchange was stored, the security analyzer can compare a) a message that was generated by the DUA during the analysis phase (possibly in response to receiving a mutated message) and b) the corresponding message that was generated by the DUA during the enumeration phase.

[0041] Note that the enumeration phase and the analysis phase are similar. In each phase, a security analyzer creates a message to be sent to the DUA. This message is then processed. In the enumeration phase, the message is processed by storing the message and its mutation points. In the analysis phase, the message is processed by mutating it according to a particular mutation. In both phases, the message is then serialized and sent to the DUA.

[0042] Also, in both the enumeration phase and the analysis phase, when a message is received from the DUA, the message is made available to be used to generate a message if desired (e.g., if the content of the next message sent to the DUA is supposed to be based on the content of the message that was received from the DUA).

II. Exemplary Implementation

[0043] When a protocol message exchange occurs between two devices (e.g., a security analyzer and a DUA), each pro-

tocol message is a byte stream. In one embodiment, a protocol message is also represented by a data structure (called a "message object") that is constructed out of various building blocks (referred to as "elements") such as structs (structures), lists, and primitives. A primitive is the most basic element and is used for integers and byte strings. A list can contain an ordered, variable number of elements (structs, primitives, and lists). A struct is a fixed-length collection of lists and/or primitives. For example, a message is represented by a message object (named "a") that is a struct that contains two elements—a primitive (named "b") and a list (named "c"):

[0044] a=Struct<<[

[0045] b=Primitive

[0046] c=List

[0047]]

In one embodiment, an element within a message object represents a field of a message. For example, element b (a primitive) within message object a (a struct) can represent a length of a message.

[0048] Also stored within a message object are one or more "mutation contexts." A mutation context includes a) an identifier of a mutation, b) a function, and c) one or more identifiers of elements within the message object. The identifier of the mutation uniquely identifies the mutation within the object. The function provides a way to mutate the object (and, therefore, the message that it represents) according to the mutation. An identifier of an element represents a field of the message that is mutated by the function.

[0049] In one embodiment, each message object supports five methods: construct, input, output, variants, and mutate. The construct method instantiates or sets up an object using any declared members and default values. The input method, which takes as input a message (e.g., a byte stream), parses the message and modifies the object to reflect the message (e.g., to reflect the values stored in the message's fields). In this way, the input method overrides the default values (which were set based on the construct method) with information obtained from a byte stream.

[0050] The output method serializes the object (e.g., into a byte stream) so that the object's data can be sent to a network or written to a file. In one embodiment, the output method operates in a cascading or recursive way. For example, consider a message object (named "a") that is a struct that contains a primitive (named "b") and a list (named "c"):

[0051] a=Struct<<[

[0052] b=Primitive

[0053] c=List

[0054]]

In order to generate a byte stream that represents the message object "a", a's output method is called. This, in turn, calls b's output method and c's output method. Note that element b and element c each support their own output methods. In one embodiment, the output method for an element is a function that specifies how to serialize that element. This element serialization can then be used to construct the serialization of the overall message object.

[0055] The variants method returns one or more "variants" and can be used to query a message object. A variant indicates a target field (as represented by an element of a message object) and a mutation to apply to that field. In one embodiment, a variant is indicated using a dotted path notation. For example, using the message object "a" described above, the variant named "a.b.mutation1" would refer to a mutation named "mutation1" that applies to the field "b" of message "a" (as represented by the element "b" within the message object "a"). In one embodiment, a variant corresponds to a mutation point.

[0056] In one embodiment, the variants method returns one or more strings (e.g., an array of one or more strings), where each string represents a variant using the dotted path notation explained above. The variants method can be implemented by, for example: Iterating through the mutation contexts of the object. For each mutation context, iterating through the identifiers of the elements within the message object. For each identifier of an element, generating a string that contains the identifier of the element followed by a period (".") followed by the identifier of the mutation.

[0057] The mutate method, which takes as input a variant, mutates a message object according to the variant. For example, calling the mutate method on the message object "a" described above using the variant "a.b.mutate1" as input would mutate the message "a" such that when a's output method is called, the resulting serialization will reflect the fact that the field "b" has been mutated according to the "mutation1" mutation. Similarly, if the message as a whole has a mutation associated with it called "mutation2" then executing the mutation a.mutation2 would mutate the message "a" such that when a's output method is called, the resulting serialization will reflect the fact that "a" has been mutated as a whole according to the "mutation2" mutation.

[0058] In one embodiment, the mutate method modifies the implementation of the output method of an element. This way, when the output method is called on the element, the element serialization that is generated will reflect the element after it has been mutated by the mutation identified within the variant. The mutate method can be implemented by, for example: Parsing the variant (which was provided as input) into an identifier of an element (e.g., "a.b") and an identifier of a mutation (e.g., "mutation1"). Identifying the element within the message object that corresponds to the element identifier (e.g., "a.b"). Identifying the mutation context within the message object that corresponds to the mutation identifier (e.g., "mutation1"). Replacing the implementation of the output method of that element with the function of the mutation context.

[0059] In this way, mutation is an alternate form of serialization. An element's default output method creates a proper (i.e., non-mutated) serialization. If the element is meant to be mutated, the default output method is overridden with a custom method (namely, the function within the mutation context). This way, when the output method is called on that element, the result will be a mutated serialization.

[0060] Thus, there is a separation between authoring a protocol message object and authoring a mutation for that object. Because of this, a mutation can be transparently applied to a protocol message object.

[0061] II.A Enumeration

[0062] Recall that in the enumeration phase, a complete message exchange is performed using proper (i.e., not mutated) messages. In this phase, some message are generated and sent to the DUA, while other messages are received from the DUA.

[0063] FIG. **2** illustrates a flowchart for a portion of the enumeration phase that concerns sending a message to the DUA, according to one embodiment of the invention. Before the method **200** of FIG. **2** begins, a non-mutated message object is generated to be sent to the DUA.

[0064] Method **200** begins: The variants method is called **210** on the message object, which queries the object and returns one or more variants. The one or more variants are received **220**. The one or more variants are stored **230**. In one embodiment, the variants are stored in a list (called a "variant list") such that the most recently identified variant is appended to the end of the list. The output method is called **240** on the message object, which returns a serialized version (e.g., a byte stream) of the message that is represented by the object. The serialized message is received **250**. The serialized message is sent **260** to the DUA.

[0065] In one embodiment, the method **200** is performed once for every non-mutated message object that is generated to be sent to the DUA.

[0066] A message can also be received from the DUA. In one embodiment, when a message (in serialized form) is received from the DUA, a message object is created that represents the received message. The message object is available to be used to generate a message if desired (e.g., if the content of the next message sent to the DUA is supposed to be based on the content of the message that was received from the DUA).

[0067] In one embodiment, the steps of the previous paragraph are performed once for every message object that is received from the DUA.

[0068] II.B Analysis

[0069] Recall that in the analysis phase, the message exchange of the enumeration phase is performed once for each variant (mutation point) that was identified during the enumeration phase. This means that once one message in an exchange has been mutated according to a variant, none of the remaining messages in the exchange will be mutated. In this phase, some message are generated and sent to the DUA, while other messages are received from the DUA.

[0070] FIG. **3** illustrates a flowchart for a portion of the analysis phase that concerns sending a message to the DUA, according to one embodiment of the invention. Before the method **300** of FIG. **3** begins, three actions are performed. These actions can be performed in any order, either sequentially or simultaneously. One, a non-mutated message object is generated to be sent to the DUA. Two, a variant is identified. This variant is the variant that will be tested by the message exchange. For example, the identified variant is a variant that was stored during the enumeration phase (e.g., during step **230** of FIG. **2**). In one embodiment, the variants are iterated through (one variant for each performance of the message exchange) in order starting with the variant that was identified first during the enumeration phase and ending with the variant that was identified last during the enumeration phase. Three, a flag is set to indicate that the variant has not yet been used to mutate a message. Recall that once one message in an exchange has been mutated according to a variant, none of the remaining messages in the exchange will be mutated. Thus, once one message has been mutated, the remaining messages can automatically be serialized and sent to the DUA.

[0071] Method **300** begins: A determination is made **310** regarding whether the variant has already been used to mutate a message. If the variant has already been used, the output method is called **320** on the message object, which returns a serialized version (e.g., a byte stream) of the message that is represented by the object. The serialized message is received **330**. The serialized message is sent **340** to the DUA.

[0072] If the variant has not yet been used, a determination is made **350** regarding whether the message object supports the variant. If the message object does not support the variant, steps **320**, **330**, and **340** are performed, as described above.

[0073] If the message object does support the variant, the mutate method is called on the message object using the variant as a parameter, which mutates the message object. The flag is set to indicate that the variant has been used to mutate a message. The output method is then called **320** on the mutated message object, which returns a serialized version (e.g., a byte stream) of the message that is represented by the object. Steps **330** and **340** are performed as described above.

[0074] In one embodiment, the method **300** is performed once for every non-mutated message object that is generated to be sent to the DUA. After the complete exchange has been performed, the exchange can be performed again (e.g., using a different variant).

[0075] A message can also be received from the DUA. In one embodiment, when a message (in serialized form) is received from the DUA, a message object is created that represents the received message. The message object is available to be used to generate a message if desired (e.g., if the content of the next message sent to the DUA is supposed to be based on the content of the message that was received from the DUA).

[0076] In one embodiment, the steps of the previous paragraph are performed once for every message object that is received from the DUA.

III. Additional Embodiments

[0077] When creating a network protocol message, the order of operations is not always as simple as top-down or bottom-up evaluation. There are many situations where length values should be computed before checksum values regardless of where these values are located in the data structure. The traditional way to solve this problem would be to attach logic to the top most element in the data structure and then perform the required computations in the correct order. The problem with this approach is that it breaks encapsulation. Network protocols have many layers, and many steps should be performed at each layer to craft a correct protocol message. For example, assume that a protocol has three layers. Layer **3** should perform a length computation on Layers **1** through **3**, and Layer **2** should perform a checksum computation on Layers **1** through **3**. If this data structure were to be evaluated top down, Layer **2** would perform the checksum and then Layer **3** would perform the length computation. This would result in an incorrectly generated packet. The correct logic would be for the checksum to be computed after the length value is computed.

[0078] In one embodiment, the computation and/or evaluation of semantic relationships like the one described above is deferred. For example, the order of operations can be defined as needed while preserving logic encapsulation. When objects are being serialized, a method can be invoked (called "after") which takes as an argument an object and a block of code. This block of code is added to a cache. When the object is evaluated, the block is then evaluated. This way, operations can be performed in any order desired while still encapsulating all the logic at the appropriate layer.

[0079] A security analyzer can be implemented in hardware, software, or a combination of both. Also, a message can be any type of structured data, such as a file. If the message is a file, the DUA would be the software application or device that opens and/or executes the file.

[0080] In the preceding description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the invention. It will be apparent, however, to one skilled in the art that the invention can be practiced without these specific details. In other instances, structures and devices are shown in block diagram form in order to avoid obscuring the invention.

[0081] Reference in the specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the invention. The appearances of the phrase "in one embodiment" in various places in the specification are not necessarily all referring to the same embodiment.

[0082] Some portions of the detailed descriptions that follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

[0083] It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise, as apparent from the following discussion, it is appreciated that throughout the description, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission, or display devices.

[0084] The present invention also relates to an apparatus for performing the operations herein. This apparatus is specially constructed for the required purposes, or it comprises a general-purpose computer selectively activated or reconfigured by a computer program stored in the computer. Such a computer program is stored in a computer readable storage medium, such as, but not limited to, any type of disk including floppy disks, optical disks, CD-ROMs, and magnetic-optical disks, read-only memories (ROMs), random access memories (RAMs), EPROMs, EEPROMs, magnetic or optical cards, or any type of media suitable for storing electronic instructions, and each coupled to a computer system bus.

[0085] The algorithms and displays presented herein are not inherently related to any particular computer or other apparatus. Various general-purpose systems are used with programs in accordance with the teachings herein, or more specialized apparatus are constructed to perform the required method steps. The required structure for a variety of these systems will appear from the description below. In addition, the present invention is not described with reference to any particular programming language. It will be appreciated that a variety of programming languages may be used to implement the teachings of the invention as described herein.

Appendix A

[0086] Data types include, for example:

[0087] ASCII (American Standard Code for Information Interchange): ascii.crlf (carriage return/line feed), ascii.cstring (character string), ascii.dsv (delimiter separated values), ascii.ipv4 (Internet Protocol version 4), ascii.length, ascii.version

[0088] ASN (Abstract Syntax Notation): asn.counter, asn.gauge, asn.id, asn.integer, asn.ip, asn.null, asn.oid, asn.sequence, asn.string.bit, asn.string.general, asn.string.ia5 (International Alphabet 5-ASCII), asn.string.octet, asn.string.printable, asn.string.utc (Coordinated Universal Time), asn.string.utf8 (8-bit Unicode Transformation Format), asn.tag, asn.timeticks

[0089] Block

[0090] Checksum: checksum.adler32 (Adler-32 checksum algorithm), checksum.crc16 (cyclic redundancy check (CRC) function using 17-bit generator polynomial), checksum.crc32 (CRC function using 33-bit generator polynomial), checksum.ipv4 (checksum of header of IPv4 packet)

[0091] Data: data.align, data.random

[0092] Encode: encode.base64, encode.qprint (Quoted-Printable (QP) format)

[0093] File: file.path

[0094] HTTP (HyperText Transfer Protocol): http.header

[0095] List

[0096] Message

[0097] Net: net.ipv4 (IPv4 address), net.ipv6 (IPv6 address), net.mac (Media Access Control address)

[0098] SMTP (Simple Mail Transfer Protocol): smtp.domain, smtp.mailbox, smtp.path

[0099] String

[0100] Type: type.count16, type.count32, type.count8, type.id16, type.id32, type.id8, type.length16, type.length32, type.length8, type.offset16, type.offset32, type.offset8, type.uint16 (unsigned 16-bit integer, type.uint32 (unsigned 32-bit integer), type.uint8 (unsigned 8-bit integer)

[0101] XDR (eXternal Data Representation): xdr.array, xdr.integer, xdr.opaque, xdr.string

Appendix B

[0102] Types of semantic rules include, for example:

[0103] Data compression algorithms—compress.zlib

[0104] Cryptographic algorithms—crypto.digest.md2, crypto.digest.md4, crypto.digest.md5, crypto.digest.ripemd160, crypto.digest.sha, crypto.digest.sha1, crypto.otp

[0105] ASCII data algorithms—data.ascii.length, data.ascii.sequence

[0106] Abstract Syntax Notation (ASN) data algorithms—data.asn.enclosure, data.asn.explicit-tag, data.asn.integer, data.asn.length, data.asn.null, data.asn.oid, data.asn.string

[0107] Checksum data algorithms—data.checksum, data.checksum.adler32, data.checksum.ipv6

[0108] Cyclic Redundancy Check (CRC) data algorithms—data.crc

[0109] Other data algorithms—data.length, data.node.count, data.offset, data.overwrite, data.pad, data.pad.roundup, data.random, data.random.node, data.segment, data.sequence, data.stacker, data.substitute

[0110] String data algorithms—data.string.int, data.string.ip, data.string.ipv6, data.string.mac, data.string.mac.multicast, data.string.mac.multicast.ipv6

[0111] Encoding algorithms—encode.base64, encode.bits, encode.eui64, encode.nbname, encode.qprint, encode.url, encode.uuencode

[0112] File algorithms—file.reader

[0113] Input algorithms—input.slurper, input.slurper.ascii, input.slurper2

[0114] Logic algorithms—predicate.exists

[0115] Protocol algorithms—protto.http.md5auth, proto.ospf.auth, proto.radius.user-pass, proto.sctp.checksum, proto.tacacsp.body

Appendix C

[0116] Types of input/output (I/O) rules include, for example:

[0117] Console—io.console, io.console.pair

[0118] File—io.file.writer

[0119] Internet Protocol (IP)—io.net.ip, io.net.ip.base, io.net.ip.pair, io.net.ip6, io.net.ip6.pair

[0120] Raw network—io.net.raw, io.net.raw.pair

[0121] Secure Shell (SSH)—io.net.ssh

[0122] Secure Sockets Layer (SSL)—io.net.ssl, io.net.ssl.pair

[0123] Transmission Control Protocol (TCP)—io.net.tcp, io.net.tcp.pair, io.net.tcp6, io.net.tcp6.pair, io.net.tcp6d, io.net.tcpd

[0124] User Datagram Protocol (UDP)—io.net.udp, io.net.udp.base, io.net.udp.pair, io.net.udp6, io.net.udp6.pair, io.net.udpd

[0125] Null—io.null.pair

[0126] Packet capture (pcap)—io.pcap.writer, io.pcap.writer.ip, io.pcap.writer.ip6, io.pcap.writer.tcp, io.pcap.writer.tcp6, io.pcap.writer.udp, io.pcap.writer.udp6

Appendix D

[0127] Mutations include, for example:

[0128] Struct::Empty—removes a sub-structure (such as a field in a message, a data item or data structure in a compound data structure, or a node or sub-graph in a graph)

[0129] Struct::Append—adds data to a structure (such as a field, message, data item, data structure, node, or graph); length and type of data can vary

[0130] Struct::Inject—injects one or more characters between child nodes; number and type of characters can vary

[0131] Struct::Overwrite—overwrites one or more characters of a graph (or sub-graph) after the graph has been transformed; number and type of characters can vary

[0132] Struct::Truncated—removes a node from a graph (or sub-graph)

[0133] Struct::Repeated—adds duplicates of one or more children of a node; number of duplicates and number of children can vary

[0134] String::EmptyLines—prepends one or more empty lines to a structure; number of lines can vary

[0135] String::Surround—adds one or more characters in order to surround one or more child nodes; number and type of characters can vary

[0136] String::Random—generates one or more random characters; number of characters can vary

[0137] String::Overflow—generates one or more characters; number and type of characters can vary

[0138] String::Format—generates a format string; type of format placeholder can vary (e.g., character string (%s), integer as hexadecimal number (%x), integer as signed decimal number (%d))

What is claimed is:

1. A method for generating an attack on an exchange of messages according to a communication protocol, comprising:
- determining a set of one or more mutation points that exist in the message exchange; and
- performing, for each mutation point in the set, a mutated message exchange.

2. The method of claim **1**, wherein determining the set of one or more mutation points that exist in the message exchange comprises:
- identifying a message that is sent to a device-under-analysis (DUA); and
- determining a set of one or more mutations that are supported by the message.

3. The method of claim **2**, wherein the message is represented by a message object.

4. The method of claim **3**, wherein the message object stores the set of one or more mutations that are supported by the message.

5. The method of claim **4**, wherein determining the set of one or more mutations that are supported by the message comprises querying the message object.

6. The method of claim **1**, wherein performing the mutated message exchange comprises:
- identifying a message that is sent to a device-under-analysis (DUA); and
- identifying, based on the mutation point, a mutation that is supported by the message; and
- mutating the message according to the mutation.

7. The method of claim **6**, further comprising sending the mutated message to the DUA.

8. The method of claim **6**, wherein the mutation point is represented by a string, and wherein identifying, based on the mutation point, the mutation that is supported by the message comprises parsing the string.

9. The method of claim **6**, wherein the message is represented by a message object.

10. The method of claim **9**, wherein the message object supports an output method that returns a serialized version of the message represented by the message object.

11. The method of claim **10**, wherein mutating the message according to the mutation comprises modifying the output method supported by the message object.

12. The method of claim **6**, further comprising:
- identifying a message that is received from the DUA; and
- creating a message object that represents the message.

13. A computer program product for generating an attack on an exchange of messages according to a communication protocol, the computer program product comprising a computer-readable medium containing computer program code for performing a method, the method comprising:
- determining a set of one or more mutation points that exist in the message exchange; and
- performing, for each mutation point in the set, a mutated message exchange.

14. An apparatus for generating an attack on an exchange of messages according to a communication protocol, comprising:
- a mutation point module configured to determine a set of one or more mutation points that exist in the message exchange; and
- a message exchange module configured to perform, for each mutation point in the set, a mutated message exchange.

* * * * *