February 25, 2016

AUTOMATED REVERSE ENGINEERING 1 - ACTIVE STATE MACHINE LEARNING

SOFTWARE TESTING AND REVERSE ENGINEERING

Roy Kokkelkoren (s1617761), Vladyslav Cherednychenko(s1750763)

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Documentnumber
3TU / EIT: Cyber Security Specialization — 201500036

UNIVERSITY OF TWENTE.

Contents

1 Introduction

The following sections contains review of papers describing particular usage of active state machine learning.

2 Literature Review

2.1 Protocol state fuzzing of TLS implementations

Active state machine learning can be used for several different applications, one of which is *protocol state fuzzing*. J. de Ruiter and E. Poll have used this application of active state machine learning to verify different TLS implementations[3].

J. de Ruiter and E. Poll have verified the implementations of the TLS[1] protocol of different software libraries using only black-box testing. Using active state machine learning it is possible to create a state model of the specific implementation. Using this approach it is possible to determine the diversity of the various TLS libraries which all implement the same protocol. The resulting model can also be used to find security flaws within the specific implementation.

The research performed by J. de Ruiter and E. Poll therefore focuses on the actual implementation of the TLS protocol, this is in contrast with previous analyses of the protocol which focused more on the abstract descriptions of TLS. The protocol state fuzzing was performed on the TLS versions 1.0 and 1.2 of the libraries. The research resulted in the discovery of several security flaws within various TLS implementations and the discovery that the internal states of the various libraries differ greatly although they implement the exact same protocol.

In order to perform the research on the TLS implementations, J. de Ruiter and E. Poll used LearnLib which is a modified version of Angluin's L* algorithm. The LearnLib algorithm requires a list of messages it can send to the SUT[2], which is known as the *input alphabet*. It also requires a command which is sued to reset the SUT to its initial state. By sending numerous messages from the *input alphabet* Learnlib will create a hypotheses for the state machine. This hypotheses is then compared to the actual state machine, if the models are not equivalent the Learnlib will use the difference to redefine its hypothesis.

Learnlib does therefore require an actual state machine, however because the tests are performed using a black-box methodology the actual state machine is not known. Therefore in order to compare the models the equivalence check has to be approximated. J. de Ruiter and E. Poll used Chow's W-method to perform this model-based testing which then is used to perform the equivalence check. J. de Ruiter and E. Poll proved to be a valuable research method to verify the implementations of protocols. have improved Chow W-method with a specific property of the TLS protocol. This greatly improved the performance required to use Chow's W-method. They used the property of the TLS protocol in which the protocol returns *Connection Closed* to every message send after a connection is closed. Improving the W-method using this property reduced the computation time using the full *input alphabet* from 4:09 hours to 0:27.

The research was performed on the TLS libraries shown in table: 1. After the research was finished there were four vulnerabilities discovered within the state models of the various TLS libraries. The vulnerabilities which were found during the research are:

- GnuTLS
  Forgetting the buffer in a heartbeat, using HeartBeatRequest messages it is possible to reset the buffer which collects all handshake messages to perform a hash over the messages to verify integrity. By also resetting the buffer at the client side it is possible to complete the handshake protocol without any integrity guarantee being met.

[1] Transport Layer Security
[2] System Under Test

- JSSE
  Accepting plaintext data, using the state model of the JSEE implementation it was discovered that there were two paths leading to the final state. It appeared that it was possible for the client to finish the handshake protocol without sending a *ChangeCipherSpec* message, which was accepted by the server. This resulted in the server expecting the client to send all the data plain-text, thereby invalidating the assumption of confidentiality and integrity. Because this behavior is transparent to applications using this connection this bug was regarded as a severe security vulnerability.

- OpenSSL
  Re-using keys, it is possible to send an additional *ChangeCipherSpec* message after the handshake was finished. This resulted in the client keys, which are used to encrypted the data which is send from the client to the server, to be set to the same keys of the server. This leads to full data compromise when on of the keys is compromised.
  Early ChangeCipherSpec, the state model of the older version of OpenSSL 1.0.1g reveals a known security vulnerability which allows an attacker to easily compute the sessions keys of the TLS session. By sending a *ChangeCipherSpec* message without a prior *ClientKeyExchange* message being send will result in the an empty master secret. The master secret is used to compute the session keys. The empty master secret therefore ensures that the session keys are calculated using public data from the session, which allows an attacker to easily calculate the session keys.

Name	Version	URL
GnuTLS	3.3.{8,12}	http://www.gnutls.org
Java Secure Socket Extension (JSSE)	1.8.0_{25,31}	http://www.oracle.com/java
mbed TLS (PolarSSL)	1.3.10	https://polarssl.org
miTLS	0.1.3	http://www.mitls.org/
RSA BSAFE for C	4.0.4	http://www.emc.com/security/rsa-bsafe.htm
RSA BSAFE for Java	6.1.1	http://www.emc.com/security/rsa-bsafe.htm
Network Security Services (NSS)	3.17.4	https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS
OpenSSL	1.0.1{g,j,l}, 1.0.2	https://www.openssl.org
nqsb-TLS	0.4.0	https://github.com/mirleft/ocaml-tls

Table 1: Tested TLS implementations

Using protocol state fuzzing proved to be a valuable research method to verify the implementations of protocols. The research discovered multiple vulnerabilities and had shown that there are varieties between the different implementations of the protocol. Using protocol state fuzzing will not discover every vulnerability and should therefore not used as the only verification method. But adding this method in a test suite will ensure that the overall functionality of the protocol will be more secure.

2.2 Inference and Analysis of Formal Models of Botnet Command and Control Protocols

Another application of active state machine learning is *protocol inference*, which is process of learning a certain protocol by either passive observation or active probing. A research performed by C.Y. Cho, et all.[2] used protocol inference to determine the state model of the C&C[3] systems of a popular botnet.

[3] Command and Control

They stated that by knowing the internal state model it would be possible to determine security flaws within the C&C systems which would make easier to perform an effective take-down of the entire botnet. They stated that identifying the critical links, design flaws, background-channels and implementation differences of the state model of the botnet would be sufficient information to perform a successful take-down.

Because the state model was generated from a live botnet certain security measures had to be implemented. C.Y. Cho, et all. therefore developed their own botnet emulators which would mimic the original functionality but will not perform any malicious action. In addition, all network traffic was send using the tor-project[4]. This, however, added a delay to every message which was send to the botnet.

In order to increase the overall efficiency of the state machine learning algorithm, even when using an anonymizing network such as tor, C.Y. Cho, et all. added additional optimization to the L* algorithm. The L* algorithm was used to generate the internal state model of the C&C systems of the botnet. By implementing multiple botnet emulators and using a parallelization and caching mechanism, it was possible to greatly increase the number of messages that could be sent to the C&C systems. The caching mechanism was the link between the sequences generated by the L* algorithm in a linear fashion which were translated into messages which could be send in parallel. The results were then interpreted by the caching mechanism to offer them in a serial method back to the L* algorithm. This decreased the overall waiting time from 6.8 to 1.4 seconds.

The L* algorithm requires a certain *input alphabet* used to actively probe the target agent. Because the protocol used by the bots to contact the C&C systems is based upon the SMTP protocol, therefore the messages available within the SMTP protocol are used as the *input alphabet*.

By using the L* algorithm it was possible to create the internal state model of the CC systems of the botnet. Using this model it was possible to determine the required information necessary to perform a successful take-down of the botnet. The researchers found the following conclusions:

- **Analyse of critical links**, the state model showed that the SMTP servers are critical in the ability to spam. In addition, the SMTP server are shared between all the bots which are in control by the botnet. Meaning that a successful take-down does not need to focus on the multiple C&C but on the few SMTP server. Taken these systems down will cripple the entire botnet.
- **Analyse of design flaws**, the botnet use templates from a template server to spam. It showed that there exist a path to retrieve the spamming template which is smaller than the default states which are executed to start spamming. Using this design flaw it is possible to retrieve new spamming templates before the botnet actually starts spamming, this in turn can be used to update spam filters.
- **Analysis of background-channels**, analyzing the state model, it shows that the template server behaves differently whether the botnet connected to the SMTP server prior to connecting the template server. This shows that there exists a background communication channel between the SMTP server and the template server.
- **Analysis of implementation differences**, as mentioned before the botnet protocol is similar to the SMTP protocol. Manual verification of the two state models of the C&C systems and a normal SMTP server, shows that there is a difference in implementation. This information can be used to implement in network-flows IDS[5] to detect any botnets which connects to the C&C systems without any false-positives on default SMTP clients.

The study performed by C.Y. Cho, et all. shows that active machine learning can be used to analyze the state model used by C&C systems of a botnet. Having access to this state model,

[4] https://www.torproject.org/

[5] Intrusion Detection Systems

can in turn be used to detect any security flaws within the system. That information can then be used to perform an effective take-down of the entire botnet.

2.3 A Model-based Approach to Security Flaw Detection of Network Protocol Implementations

Using active state machine learning is a popular approach on determining the security status of an implemented protocol. This is because testing the implementation of a protocol is difficult. Y. Hsu, et all.[4] performed another research of validating the implementation of a protocol, in this case the MSNIN[6].

There currently exists multiple approach in testing protocol implementations. But as Y. Hsu, et all. discussed these approaches are quite limited because of the *black-box implementation* and *lack of formal specification*. In addition, current testing approaches such as *random detection* or *syntax based flow detection* do not take the formal behavior model into account when performing the tests. This greatly the decreases the overall fault coverage of the implementation. Y. Hsu, et all. therefore created a new method, which uses active state machine learning, called *Formal Model Based Flow Detection* that increases the fault coverage to also include the Formal Behavioral Model.

Formal Model Based Flow Detection works by performing fuzzing tests following a certain flow within the implementation of the protocol. The flow which is performed by the fuzzing test is determined using the FSM[7] model of the implementation. Therefore calculating the FMS model is one of the key steps of perform a Formal Model Based Flow Detection test. The model is computed using the L* algorithm. As mentioned before the L* algorithm requires a set of messages which are used to actively probe the agent. In case of the research performed by Y. Hsu, et all. the protocol which is investigated is the MSNIN. This protocol is publicly known and therefore it is possible to gather valid messages for the protocol and use that as the input alphabet.

The actual fault testing is performed by using a synthesized FSM model to guide multiple fuzzing functions. These functions send different alterations of messages depending on the individual function. The overall objective of these functions is to verify if certain inputs result in the *GOAL* state. The *GOAL* predicate verifies if a certain output contains signs of a *'crash'*, in this case *'crash'* is equal to an *'error'*. Using the FSM model, the researchers were able to develop 26 single message fuzzing functions, which can be categorized using four categories:

- Data Field Fuzzing
- Message Type Fuzzing
- Intra-Session Message Reordering
- Transition Substitution

Y. Hsu, et all. performed the Formal Model Based Flow detection test on two open source clients of the MSNIN protocol, namely aMSN and Gaim (now known as Pidgin). By using there new model testing approach they discovered that almost all fuzzing functions resulted in a crash of the protocol implementation. This shows that the client applications contain faulty implementations which could lead to uncertain states which in turn could lead to security vulnerabilities.

The new Formal Model Based Flow detection was compared with a previously mentioned testing approach. The results shows that the newly developed testing approach resulted in a higher detection of crash instances and the previous testing methods, this shows that using a FSM model increases the fault coverage. In addition, by ensuring that the synthesized FSM models is 100% complete it is possible to ensure that a 100% complete fault coverage will be fulfilled.

[6] MSN Instant Messaging
[7] Finite State Machine

The method developed by Y. Hsu, et all. shows that fuzzing tests using *random detection* or *syntax based flow detection* does not have a high flaw coverage. Using the Formal Model Based Flow detection will result in a higher flaw coverage.

2.4 Analyzing program behavior through active automata learning

In this summary we are going to describe a method used by authors of [1] to tackle the problems of RERS Challenge 2013. Despite impressive progress in automated program analysis there are many obstacles that researchers have to overcome. Generally, program analysis is divided into three categories:

- **Black-box**, when only binary of a program is given
- **White-box**, when the full source code of program is given
- **Grey-box**, when previous two are combined in different ways

While classical program analysis works on the level of a programs state space as constituted by heap contents, local variables etc., active automata learning drastically narrows this perspective, merely focusing on the observable (I/O) behavior. Aim of active automata learning is to infer a finite-state model of a given programs observable behavior. Interface to the program should be given as a starting point. Active automata learning can also be adapted to grey-box and dynamic white-box analysis methods.

Preliminaries

Interaction with System Under Learning (SUL) starts by executing a number of actions on a system and observing the reaction to each action. The actions set is called an input alphabet and it should be finite. If a set of possible actions that can be invoked on a system is infinite (what happens in a lot of cases) then a suitable abstraction on the input alphabet has to be defined either manually or semi-automatically. Each action has to be executed separately, so the observed behavior does not depend on internal state of a SUL.

The biggest limitation of active automata learning is its restriction to deterministic systems. But determinism of a system is a very relative term and can be achieved through a suitable abstraction.

The learning process can be divided into two phases:

1. Exploration phase, where sequences of input symbols (membership queries) are executed and their outputs are analyzed. The result of this phase is a conjectured hypothesis modeling the SULs behavior.
2. Validation phase, where the conjectured hypothesis from the previous state is checked for completeness and correctness with respect to systems behavior. This process realizes the so-called equivalence query.

Obviously, the validation requires full knowledge about the SUL, so in practice it is usually approached by conformance testing. If the model is found to be incorrect, the equivalence query yields a counterexample which can be used to refine the hypothesis. Otherwise, the learning is stopped.

We will briefly run through the algorithm that authors of the paper used in their approach. So the outputs of membership queries are organized in an observation table which is vertically divided into two parts: lower and upper. Both parts are labeled with words (prefixes) over the input alphabet. Prefixes in the upper part are called short prefixes (S_p), prefixes in the lower part long prefixes (L_p).

S_p serve as unique representatives for the unknown states in the target system. L_p are the one-letter extensions to those in the upper part and correspond to transitions in the hypothesis.

Columns in the observation table have the purpose of distinguishing states in the target system and are called suffixes (*D*). A cell contains the outcome of executing the concatenation of a prefix (row label) and a suffix (column label) on the system, reduced to the part attributable to the suffix. The contents of a row can be seen as a partial residual finitely approximating the Nerode relation. It is used to distinguish rows in the upper part (states), and match transitions from the lower part with a unique target state.

After the first iteration we have an initial state of the system as a short prefix, a predefined set of suffixes is used to characterize this state. If a transition reveals previously unseen behavior with respect to a certain suffix, a new state has been discovered. Consequently, the corresponding row is moved to the upper part of the tables, and the long prefixes will be augmented by all one-letter extensions of the corresponding prefix. Otherwise, the table is called closed, and we can use it to construct a hypothesis.

The counterexample mentioned before is a word w for which the output of a real system differs from the output predicted by the hypothesis. It gives a rise to at least one additional state in the hypothesis.

LTL model checking

Model checking is an automated technique of verifying that a formal model of a system conforms to a given specification, often given in temporal logics such as LTL or CTL. In the paper being summarized, a nondeterministic Bchi automation is used for model checking. For details please refer to [1].

Tackling the challenge problems

The RERS challenge task was to evaluate the validity of a set of 100 LTL formulae and additionally the reachability of 60 designated error labels for each of the 27 problems in every category (black-box, white-box, grey-box).

Black-box scenario

Problems are provided as 27 executable binaries $P_1 \dots P_{27}$ to answer the challenge tasks. They are divided into three categories, where the input alphabets are sets of 6, 10, and 20 integer values, respectively. Beside the set of problems and their corresponding set of inputs, each problem is provided with additional information of approximately 105 execution traces consisting of already observed input/output behavior which we will refer to as the passive data.

In the original learning setup, the authors used the algorithm described above. The major limiting factor of their approach was that the potential quiescence (when executing an input causes no output) of programs required an adequate timeout. In other words, in order to be sure that the learning algorithm correctly recognizes the quiescence, the significant timeout is needed between executions. As a consequence, experiments were running for hours or even days without finishing.

Equivalence queries were approached with a three-step method. In the first step, random sampling of words was performed, with parameters chosen depending on the current hypothesis: they sampled up to 2 |Q| $|\Sigma|$ words with a length uniformly distributed between |Q| and 2|Q|.

In the second step, after the random sampling returned no further counterexamples, a W-method conformance test with a depth parameter of two was conducted on the hypothesis model. An implementation for the W-method constructs traces that are generated through the concatenation of all words from a transition cover, all words of length not more than depth and words from a characterizing set of the hypothesis automaton. Then all of these words

were validated, e.g. the output predicted by the hypothesis is correct with respect to the output observed from the target system.

Finally, in the third step, the passive data gets involved and was used to refine the hypothesis whenever one of the provided traces was a counterexample.

White-box scenario

In the white-box scenario authors modified each programs code without changing its semantics. The output was stored in variable instead of printing it to the console. Then they simply called method *calculateOutput(input)* for each symbol and collected outputs from it.

2.5 Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution

Protocol reverse engineering is a manual work in most cases and takes a lot of time. Complexity of this work arises due to following reasons:

1. Protocol message contains a lot of fields.
2. An individual field can be of various size.
3. Complex relationships between fields may exist.

Another approach to make protocol RE easier called AutoFormat is presented in the paper being analyzed. The aim of this protocol is to uncover both fields of a protocol message and relations between them. It works by analyzing adjacent message bytes that are handed in the same execution context. AutoFormat is designed mostly for unknown protocols, because source code of protocol is not needed for it.

Problem and terminology

Two main problems in protocol RE are following:

1. Identifying the boundary (or size) of every protocol field and the entire structure of fields.
2. Build entire protocol state machine.

Another problem is that a protocol field can contain multiple sub-fields.

Finest-grained field is a smallest subsequence that cannot be further divided into sub-fields. Further on, we will denote a field in a protocol format as ϕ(x). Also three types of relations between fields are defined:

1. Protocol message contains a lot of fields.
2. An individual field can be of various size.
3. Complex relationships between fields may exist.

System Design

Architecture of AutoFormat has 2 main elements:

- Context-aware execution monitor
- Protocol field identifier

Given a binary that implements the protocol to be analyzed, AutoFormat works like following:

1. On receiving an incoming protocol message, it first marks the received data and keeps track of their propagation at the byte granularity.
2. Once a message byte is read, the execution monitor logs that particular byte, its offset in the entire message, and the run-time execution context at that moment, which includes the call stack and the location of the instruction being executed.
3. With the collected context information, the off- line protocol field identifier is invoked to identify protocol fields and extract the structural layout of the message.

Context-Aware Execution Monitor:

While monitoring program execution, the network-related system calls can be intercepted, messages can be marked as received and every message byte can be annotated with its offset in the entire message. Furthermore, data movement instructions (e.g. mov) are also instrumented as well as arithmetic/logic instructions. For more specific details on data movement operations refer to [6].

AutoFormat is interested in two types of execution context information:

- the run-time call stack and
- the address of the instruction that accesses a marked memory location

Run-time call stack information is acquired by traversing stack frames, but this technique highly depends on fact that program was compiled with debug symbols included. This problem can be solved by instrumenting the function call and return instructions and maintaining a shadow stack frame inside the execution monitor.

Protocol Field Identifier

When a marked memory location is being read, AutoFormat will log the execution context and save it as a record in the form of $< o, c, s, l >$, where o is the offset of the referenced memory in the entire message, c is its content, s is the run-time call stack when the memory reference occurs, and l is the location of the memory reference instruction. Log file is processed as an array (denoted as log) with N elements in it. Each element has four members: *log[i].o, log[i].c, log[i].s*, and *log[i].l*. In practice log file is preprocessed to remove identical entries.

Next step is to build a protocol field tree (textitftree) and store the identified fields there. Each node of textitftree represents either a finest-grained field or a hierarchical field. Each field is associated with an offset interval denoted by the starting position and the size of the field.

Essentially, algorithm scans the entire log file and checks whether two successive records (*log[i]* and *log[i1]*) are related to two consecutive offsets of the input data and have the same execution context. If so, it merges the corresponding offset intervals into one. If not, a new protocol field node will be created. To link the new node to the tree, an existing node will be chosen as the new nodes parent node. This chosen node but not its child (if any) should contain the new nodes offset(s). If such a node cannot be found, it will insert the new node as a child of the ROOT node which contains all offsets. For the parent of the new node, some of its children may be moved down to become the new nodes children. For more details and code refer to **Algorithm 1** in [6].

The *ftree* built by the above algorithm can have some issues, like:

- some leaf nodes might be of overly fine granularity
- certain fields may be referenced multiple times at different time instances
- there exist some fields that may not be referenced at all

For these types of issues, following solutions are proposed:

- **Tokenization:** merging two nodes under certain conditions.
- **Redundant node selection:** iteratively removing redundant nodes.
- **New node insertion:** insertion of a new child node with the missing offsets.

Identifying Parallel and Sequential Fields

For parallel fields, the algorithm collects the execution history seen by the lowest offset of each node in the *ftree*. For a parent node, if some of its child nodes share similar execution history (share common history prefix), it will mark each of them as a parallel field. If there exist non-marked child node(s) between two marked ones, the marked one on the left (with a smaller offset) will join the non-marked child node(s) to form a new parallel field.

For sequential fields, AutoFormat first performs a pre-order traversal of the tree but only lists the leaf nodes and those internal nodes that each represents a parent of multiple parallel fields. The result of this traversal is a list of sequential fields. Recursively, the same traversal is performed on the sub-trees each rooted at a hierarchical node that represents a parallel field. This way it is able to identify all lists of sequential fields in the protocol field tree.

2.6 A Taxonomy of Botnet Behavior, Detection and Defense

This paper[5] presents, to the best of our knowledge, the first systematic analysis of the botnet threat from three aspects:

- botnet behaviors/architectures,
- detection mechanisms, and
- defense strategies

The first one classifies botnets on their behavior: host infection, rallying, command and control (C&C). The second taxonomy classifies different approaches to botnet detection. The third taxonomy classifies botnet defense strategies.

Taxonomy of botnet behavior

In this taxonomy, botnet behavioral features are categorized as those concerning *Propagation, Rallying, C&C, Purpose and Evasion*.

Propagation Depending on the degree of required human intervention, propagation mechanisms can be broadly classified as active and passive.

1. *Active:* In this mode of propagation, the botnet is capable of locating and infecting other hosts without any (human) user intervention.
2. *Passive:* Passive propagation requires some level of user intervention. Three most widely used passive propagation techniques are:
    - Drive-by download. Propagate through infected web sites.
    - Infected media. Propagate through media drives: USB sticks, hard drives, etc.
    - Social engineering.

Rallying mechanism Rallying is the process used by bots to discover their C&C servers. Here are some commonly used rallying methods:

1. IP address: can be hardcoded or dynamically assigned.
2. Domain name: domain is used as a link to facilitate communication with C&C server. Can be either hardcoded to bot binary or generated using domain generation algorithm.

Command and control C&C communication can either leverage existing communication protocols (IRC, HTTP, Gnutella) or use custom-made protocols for this purpose. Usage of an existing algorithm is more reliable, but easier to classify.

Purpose The main purpose of botnet is to use its combine power to carry out malicious activities on its behalf. But there are some prime purposes that are being served by botnets, they are:

1. Information gathering
2. Distributed computing
3. Cyberfraud
4. Spreading malware
5. Cyberwarfare
6. Unsolicited marketing
7. DDoS attacks

Evasion Botnets operate stealthily to evade detection and increase their probability and duration of survival. We can view the evasion strategies adopted by a botnet from the perspective of the bot, botmaster, C&C server and C&C communication.

From bots point of view, a number of mechanisms are employed to remain alive. Binary obfuscation is widely used to avoid pattern-based detection. To evade honeypots, some bot binaries perform some checks on environment they are being executed in. Some botnets may disable available security software on a machine after infecting it. Furthermore, bot binary may work as a rootkit which makes it invulnerable to anti-virus software.

C&C servers is the most important part of the botnet, so a lot of attention is paid to evasion techniques for them. One technique is called *IP Flux*. Its idea is that the IP address of the domain name is frequently changed.

Another technique is called *Domain Flux* it associates multiple domain names with same IP address. Also the DNS server itself can be located in one of the countries that are indifferent to requests to take down malicious services. Moreover, anonymization techniques are used to hide a sender of a message.

Communication between C&C server and bots is hidden using following techniques:

1. Encryption
2. Protocol manipulation (HTTP, IPv6 tunneling)
3. Traffic manipulation
4. Novel communication techniques

Topology Topology of a botnet can be either centralized or decentralized. Botnets that follow centralized topology are managed by a single C&C server. Centralized topology can be further separated to *star* (fast in terms of communication, but entire botnet goes down if C&C server is unavailable) and *hierarchical* (multiple layers are incorporated between botmaster and bots) types. In decentralized topology no single entity is responsible for providing command and control to bots. Bot management is either distributed among multiple C&C servers or there is no obvious master-slave relationship between bots and C&C server.

Taxonomy of botnet detection mechanisms

Based on which component is being targeted, botnet detection is classified into different facets: bot detection, C&C detection and botmaster detection. Before going further, some terms should be defined to make everything clear.

Definition 1. *Botnet Detection: Detection of all components of a botnet, comprising the botmaster, C&C server(s), means of C&C, and (all or a subset of) bots.*

Definition 2. *Bot Detection: Detection of botnet infected machines, with or without regard to bot families.*

Definition 3. *Bot Family Detection: A class of Bot Detection focused on bot family detection.*

Bot detection

The bot detection techniques can be separated into active and passive detection. Active bot detection involves participating in the botnet operation. A defender-controlled machine can act as a bot to gain details about other bots. The other way is to hijack the C&C server by exploiting rallying mechanism, physically seizing server or by redirecting all communication traffic to defenders machine.

Passive detection approaches detect botnets by silently observing and analyzing their activities without making a conscious effort to participate in the proceedings. Passive techniques can be *Syntactic* (pattern-recognition) or *Semantic* (detecting deviations in behavior).

C&C detection

Identification of C&C and its subsequent analysis can help in understanding botnet behavior. It can also be passive or active.

Active C&C detection involves taking part in the botnet operation, for example, online manipulation of network flows to deduce information about possible C&C communication. Active detection involves injection and suppression. Injection entails injecting packets into suspicious network flows, where the similarity of the reply to the injected packets with typical bot response indicates that the flow might be part of C&C communication. In suppression, incoming/outgoing packets in suspicious network flows are suppressed to elicit known response from any of the ends of the C&C communication.

In passive detection involves syntactic and semantic methods. Syntactic C&C detection works by developing signature-based models of C&C traffic. The signatures are obtained by observing frequently occurring strings or token sequences in malicious traffic. Semantic C&C detection approaches use some heuristic to associate certain behavior with C&C traffic. These can be further divided into Statistical, Correlation and Behavior-based approaches.

Taxonomy of botnet defense mechanisms

At present, defense against botnets is mostly preventive or defensive. Preventive defense includes proactive measures to avoid botnet infection. Defensive methods are reactive in na-

ture and concern themselves with cleaning systems once they have been infected

Botnet defensive mechanisms can be divided into two categories - Preventive and Remedial. Preventive mechanisms are:

1. Technical: they include *host cleaniness* (checking for bot binaries) and *network cleanliness* (NANS, etc.)
2. Non-technical: *attacker dissuasion, legal accountability, user education*

Remedial mechanisms are:

1. Defensive strategies: *host-based* (recover the machine after botnet infection) or *network-based* (secure the network after all hosts are recovered)
2. Offensive strategies: *indirect attack* (reduces usability of the botnet to the botmaster) or *direct attack* (entails mechanisms to di- rectly hurt one or more components of the botnet)

Botnet defense comprises mechanisms that help in prevention of botnet infection. This is not always a dependable option as botnets represent advanced threat and the intrusion may succeed despite proactive security measures. Botnet defense also covers reactive techniques such as recovery from botnet infection or offensive measures to disrupt or impair the botnet.

3 References

References

[1] O. Bauer, M. Geske, and M. Isberner. Analyzing program behavior through active automata learning. *International Journal on Software Tools for Technology Transfer*, 16(5):531–542, 2014. cited By 1.

[2] C.Y. Cho, D. Babi, C.E.R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. pages 426–439, 2010. cited By 38.

[3] J. de Ruiter and E. Poll. Protocol state fuzzing of tls implementations. 2015.

[4] Y. Hsu, G. Shu, and D. Lee. Model-based approach to security flaw detection of network protocol implementations. pages 114–123, 2008. cited By 18.

[5] S. Khattak, N.R. Ramay, K.R. Khan, A.A. Syed, and S.A. Khayam. A taxonomy of botnet behavior, detection, and defense. *IEEE Communications Surveys and Tutorials*, 16(2):898–924, 2014. cited By 6.

[6] L. Zhiqiang, J. Xuxian, X. Dongyan, and Xiangyu Z. Automatic protocol format reverse engineering through context-aware monitored execution. 2008.