

The HoneyTank : a scalable approach to collect malicious Internet traffic

Nicolas Vanderavero, Xavier Brouckaert*, Olivier Bonaventure, Baudouin Le Charlier

Abstract—During the last few years, the amount of malicious traffic on the Internet has increased due to the spreading of worms, various port scanning activities, intrusion attempts or spammers. Collecting and analyzing this malicious traffic is an important issue. It can teach us what are the latest trends in computer misuse, it can help us discovering new kinds of attacks or it can be used to automatically generate signatures for network-based intrusion detection systems. In this paper, we propose an efficient method for collecting large amounts of malicious traffic running over TCP. The key advantage of our method is that it does not need to maintain any state to emulate TCP services running on a large number of emulated end-systems. We implemented a prototype on the ASAX IDS and provide in this paper several examples of the malicious activities which were collected on a campus network attached to the Internet. We explain how we implemented various protocols in a stateless way and we discuss limitations of our approach. We also discuss how our method can be improved to make an accurate but still stateless emulation of stateful protocols.

Index Terms—Honeypots, Intrusion Detection Systems, Worms

I. INTRODUCTION

THE Internet is not anymore a research network used only by researchers and grad students. The TCP/IP protocol suite is now ubiquitous and the Internet is used to carry various types of information, from large scientific datasets to Voice or Video over IP. Unfortunately, the Internet is also being more and more used for various malicious activities such as intrusion attempts, phishing [1], denial-of-service attacks, spamming, worms [2], [3] . . . and some automated attacks can compromise a large number of computers in a short period of time. Most experts indicate that the amount of illegitimate traffic on the Internet is growing. Faced with those problems, it is important to be able to obtain more informations about the characteristics of the malicious traffic on the Internet.

A first solution to detect and collect malicious traffic is to monitor the IP packets passing on the production links of a network and use packet analysis tools to determine the infected machines based on signatures. Some of the deployed solutions use a network-based intrusion detection system like Snort [4] : they capture all the packets including their payload and use rules to identify infected machines.

This solution is often used by enterprises, but it suffers from two problems.

The authors are within the Computing Science and Engineering Department, Université catholique de Louvain (UCL), Belgium, <http://www.info.ucl.ac.be>, email: {nv, xbr, Bonaventure, blc}@info.ucl.ac.be

*Supported by a grant from FRiA (Fonds pour la Formation la Recherche dans l'Industrie et l'Agriculture, Rue d'Egmont 5, B-1000 Bruxelles, Belgium).

First, it cannot be deployed everywhere and thus it only monitors the traffic on a small number of links, typically the Internet access links of corporate networks. This means that a worm spreading first on the local subnets [5], [6] will be detected lately, i.e. when the corporate local subnetworks are already infected. Second, as the network link carries both legit and malicious traffic, analysing it in real-time on a high bandwidth link can be difficult. False-positives alerts may be generated as it is sometimes necessary to lower the complexity of the detection rules to achieve real-time analysis.

In addition to monitoring the traffic exchanged on a network link, we can learn a lot from monitoring the traffic sent to unused addresses of a network. Since no devices are connected to those addresses, nobody should try to contact them (typo errors and misconfigurations excepted). Thus, every request sent to an unused address should be considered as suspect. Moreover, the bandwidth used by this traffic should be significantly lower than the 'normal' traffic. It should thus be easier to analyze it in real-time.

Collecting the traffic sent to unused addresses has been used in other projects like [7]–[9]. These related works are discussed in section II of this paper.

As TCP is the protocol used to transport more than 90% of the total Internet traffic [10], it is important to accurately measure the malicious traffic running over TCP. This paper presents the design and the experimentation results of what we call a HoneyTank. A HoneyTank is a workstation receiving TCP segments sent to unused or unallocated IP addresses and replying to those segments to emulate real end-systems supporting real TCP services. Given the amount of segments that could be collected by such a system during the spreading phase of a worm, it must be able to handle a huge amount of simultaneous TCP connections. To achieve this, we propose a HoneyTank that emulates TCP services without maintaining any state and report our experience with a prototype based on the ASAX IDS.

The remainder of this paper is organised as follows. The section II discusses the related works. In section III we discuss several methods to collect malicious traffic in operational IP networks. In section IV we describe our prototype HoneyTank implemented on the ASAX IDS. Section IV analyses the performance of our prototype and we describe in section V the malicious traffic captured on a campus network. We identify the limitations of the HoneyTank approach in section VI.

II. RELATED WORK

Various methods exist to collect traffic sent to unused IP addresses. A first approach is to configure the routers to export

flow-information [11] to a monitoring station and use this information to detect the infected machines [7]. A flow record indicates for each layer-4 flow passing through the router the IP source, IP destination addresses, the source and destination ports, the transport protocol (UDP/TCP), ... By processing this information, the monitoring station can detect which machines are currently scanning the network and could take actions to block the source of this traffic [12]. However, blocking a suspicious machine after the arrival of TCP SYN segments from its IP address is not always a good idea since an infected machine sending spoofed packets could easily deny service to legitimate hosts. Instead of flow-information, firewall logs [13], [14] could be also used.

Two methods for collecting malicious traffic have been proposed in [8] and [15], [16] : Darknets and "network telescopes". They use a monitoring station running a network sniffer and a default network route announced toward this station. When a packet is sent to an unallocated subnet of the network, it will take the default route and will be analyzed by the monitoring station. However, Darknets and network telescopes usually do not respond to the received packets. This implies that in the case of malicious traffic sent over TCP, a network telescope will only collect the TCP SYN segments. This is not sufficient to determine precisely the type of malicious packets received. Since those methods do not reply to incoming connections, they are limited to the analysis of ICMP and UDP traffic or TCP port scans.

If we want more accurate data, we have to reply to the connection requests. For this, a low-interaction honeypot such as Honeyd [17] can be used. Honeyd is a program which can fake the behavior of many TCP/IP stacks implementations. A complete network topology can be simulated by Honeyd, including routing, link latency or bandwidth. It handles by itself ICMP messages and the opening and clearing of TCP connections. The emulated services are not provided by Honeyd. They must be implemented either as external programs which are spawned by Honeyd on each TCP connection or as Python modules interpreted by the Python interpreter linked with Honeyd. Honeyd can also intercept network system calls of standard applications and virtualize their execution inside the honeypot. Our HoneyTank could have been implemented as a Python module for Honeyd.

Compared with Honeyd, an advantage of our HoneyTank is that it emulates the services without maintaining any state and thus can support a potentially unlimited number of addresses and ports. Yet, a disadvantage of the HoneyTank is the lower accuracy of emulated services. A human attacker could easily detect that he is connected to a fake service, but we believe that it is not a severe problem when collecting automated malicious activities such as worms.

Another method has been proposed in [9]. The iSink architecture is composed of three parts. The first is a passive monitoring component analysing flow-information. The second part is an active component based on a VMWare Honeynet. The last is an active component called the Active Sink. This component is very similar to our HoneyTank. It contains a set of stateless responders for various protocols (HTTP, Windows RPC Service, ...) implemented in Click [18].

Click is a modular software router. The user can combine together various predefined modules which process network packets. The user can also write its own modules in C++. Once combined, the modules can be run in kernel space under Linux to achieve maximum efficiency.

Another approach to reply to connection requests is to use high-interaction honeypots [19] (that is, real computers intended to be compromised). A solution using a virtual network of three machines built on top of VMWare has been described in [20] and in [21]. The drawbacks of this approach are that usually, only a small number of IP addresses are used and exposed, and managing several machines is not always trivial. This solution is complementary to our approach. It is useful to monitor the network activity off a few IP addresses in the long run. Our approach is rather useful to monitor large subnets and can quickly collect a large amount of malicious traffic.

By replying to incoming requests, our HoneyTank can be considered as an "active" method for collecting malicious traffic. If we are interested in worms detection, a more passive approach has been proposed in [22]. Since most worms scan the Internet randomly, an infected host will try to contact more unused IP addresses than non-infected hosts. Thus an infected host will receive many ICMP destination unreachable messages. They propose a solution where routers generate a duplicate ICMP destination unreachable message and forward them to a monitoring station. By analyzing those messages, they can detect hosts having an abnormal behavior. A HoneyTank attached to a default route inside an ISP as described in section III would collect much more information than only the offending IP addresses.

One possible use of the collected traffic would be to analyze it with Honeycomb [23]. This tool uses protocol analysis and pattern-detection methods on the captured traffic and can generate signatures for network intrusion detection systems. Our approach monitors a large range of IP addresses and can gather a vast amount of malicious traffic in a short period of time. Used together with Honeycomb, this could allow to quickly generate detection rules for new attacks.

III. COLLECTING MALICIOUS TRAFFIC

In this section, we discuss how the ISP's routers and enterprises or campus networks should be configured to allow a HoneyTank to collect malicious traffic.

A. ISP networks

Several ISPs, and in particular those providing broadband access are now trying to limit the spreading of worms as an added value service to their customers. One method is to use access-lists on the routers to block IP packets on ports used by worms. This approach is possible for some ports, but not for common ports like 80 (http). Monitoring all the ISP's traffic to track worms is also difficult given the amount of data to be processed.

If the routers of the ISP maintain full BGP routing tables, a simple solution to capture worm traffic is possible. At the time of this writing, only 70 % of the IPv4 addresses are

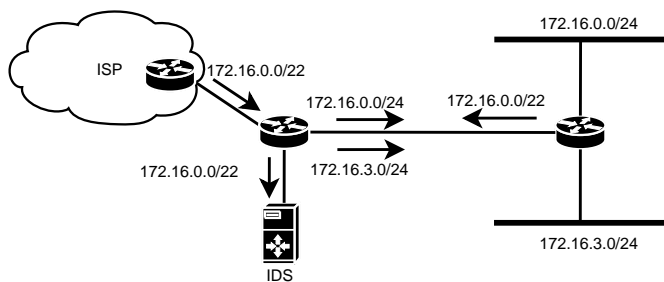


Fig. 1. Collecting packets in a campus network

announced in the global Internet [24]. When a router receives a packet, it consults its forwarding table and sends the packet along the route with the most specific prefix corresponding to its destination address. Thus, a simple solution to collect the packets sent to unadvertised IP addresses is to advertise, inside the ISP, a default ($0.0.0.0/0$) route toward the monitoring station. By tagging this route with the BGP NO_EXPORT community, it will not be advertised outside the ISP.

B. Campus and enterprise networks

In enterprise or campus networks, the spreading of worms needs to be tackled in two ways. First, the enterprise must be able to detect the external addresses that are trying to infect the enterprise network. Once such an address has been found, the typical outcome is to configure the firewall to completely block any access of this address to the campus network. Second, the campus network must be able to detect worms spreading inside the campus. As worms often start to scan in the local subnet before trying distant addresses [5], [6], it is important to quickly detect those local scans.

The routers of a campus network usually do not maintain full BGP routing tables and the solution described above for ISPs is not applicable here. In many campus networks, the allocated block of IPv4 addresses is large and some subnets of this block are not used in the campus. Consider a campus that obtained the $172.16.0.0/22$ block from its ISP. Inside this block, the campus currently uses $172.16.0.0/24$ and $172.16.3.0/24$. To collect malicious traffic, the campus could configure one of its routers to route $172.16.0.0/22$ to the monitoring station and advertise this route in its IGP (figure 1). As $172.16.0.0/22$ is also advertised by the campus ISP, the monitoring station will collect all packets sent to the IPv4 addresses in the $172.16.1.0-172.16.2.255$ range. When the campus network needs to use a part of those subnets, it simply advertises the new prefix inside its IGP and the packets sent to those addresses will reach their destination.

To detect the local scans, the IDS would need to be present on all subnets. A simple solution to reach this goal would be to reserve on each subnet an IPv4 address for the IDS and configure each router to perform proxy ARP for this address and tunnel the packets sent to this address via a GRE tunnel to the IDS [17]. Unfortunately, deploying such a solution could be difficult from an operational viewpoint as each router needs to be manually configured to perform proxy ARP.

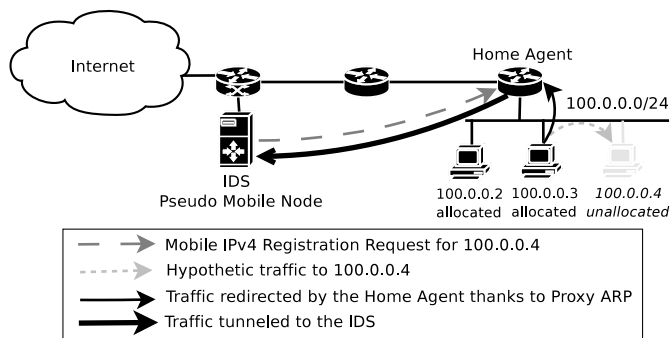


Fig. 2. Using Mobile-IP to redirect traffic to the IDS

As campus networks often maintain a list of all the allocated IP addresses in the campus either on their DNS servers or through their DHCP servers, a better solution would be to use this information to determine the unused IP addresses and to automatically configure the routers to collect the packets sent to those addresses. This can be achieved by using Mobile IPv4 which is now supported by recent routers.

Mobile IPv4 [25] (MIPv4) enables a node to move across subnets while maintaining its IP connections alive. In the simplest case, the node sends a Registration Request to its Home Agent with its Co-located Care-of Address (an address valid on the foreign network) when it arrives on a foreign network. The Home Agent then forwards packets sent to the home address of the mobile node to its current location using IP-in-IP tunnels. On the home link, it also automatically performs proxy ARP.

We can use Mobile IPv4 with another aim : redirecting malicious traffic toward the IDS. The key is to let the IDS act as a multitude of fake mobile nodes having the unallocated addresses as home addresses (figure 2).

Let us suppose that the IDS knows from the DNS or DHCP server the list of unallocated addresses in the internal network. We want to let the IDS receive all traffic sent to unallocated addresses in all subnets. First, we enable the Home Agent functionality in the routers connected to subnets we want to observe. Then we create the necessary security associations for all unallocated addresses in the Home Agent. A basic MIPv4 security association consists in a Security Parameter Index (SPI) and a MD5 key. If we want to reduce the configuration burden, we can choose to always use the same security association for all unallocated addresses. The MIPv4 control traffic can be limited by using long lifetimes for the MIPv4 Registration Requests. If an unused address needs to be allocated, the IDS simply sends a Registration Request with a zero lifetime to cancel the previous registration and to stop traffic redirection.

We have verified in a lab environment that this solution works. The routers used were Cisco 3640 with IOS 12.2(7). We simulated MIPv4 Registration Requests using Scapy [26]. Our IDS running ASAX could afterwards answer to traffic sent toward unused addresses on the Home LAN.

IV. A PROTOTYPE HONEYTANK IN ASAX

In order to analyze and respond to redirected requests, we use the ASAX intrusion detection system [27], [28]. We provide a short overview of ASAX (Advanced Sequential Analyzer on uniX) in the next section. A detailed description of ASAX can be found in [29].

A. Overview of ASAX

ASAX is a generic system designed to efficiently analyze sequential files like security audit trails. Every specific instance of ASAX is composed of three parts : the analyzer, the rule declarations and the format adaptor.

The format adaptor reads the data to analyze and converts it to the internal representation format of ASAX called Normalized Audit Data Format (NADF). This way, it is easy to modify ASAX to support a new kind of file format : the analyzer is left untouched and we only need to write a new format adaptor for this format. In our implementation, we wrote a format adaptor based on `libpcap` [30] to let ASAX analyze network traffic.

The analyzer receives its input from the format adaptor and analyzes it according to the rule declarations. To be efficient, the ASAX analyzer applies the whole set of rule declarations to all data to be analyzed, in parallel. Therefore, the analysis process is done in one pass on the data.

Rule declarations are written in a rule-based language called RUSSEL. A rule declaration can be seen as the code of a (lightweight) thread, which is responsible for analyzing a single record and take actions regarding the content of the record. The actions can be logging information about the record, sending an alarm, triggering the execution of a new rule instance for the next or current record, ... A rule-based language is convenient to write a program which searches for a particular pattern of records in a sequential file, or to verify if a sequence of events matches a given finite state machine. Furthermore, a wrapping mechanism allows the set of actions available in RUSSEL to be extended by linking C functions to the analyzer. In our implementation, we extend RUSSEL by including C-functions based on `libnet` [31] to be able to reply to the network requests.

The analysis process is done as follows : ASAX maintains a set of rule instances to be executed for the current record (called the *for_current set*) and a set of rule instances to be executed for the next record (the *for_next set*). The analyzer receives a NADF record from the format adaptor. In our case, it is a network datagram translated into NADF. Then all rule instances in the *for_current set* are executed against this record. The rules can take various actions. Once the *for_current set* is empty, the rules in the *for_next set* are moved to the *for_current set*, a new NADF record is fetched and the analysis goes on.

ASAX and its RUSSEL language are particularly well fitted for writing stateful detection rules because rule declarations can have parameters, which can be instantiated to keep relevant information about previously analyzed records. Moreover, the analyzer is able to handle thousands of rule instances at the same time. Nevertheless, the HoneyTank only uses stateless rules to achieve maximum scalability.

B. Emulation of the TCP three-way handshake

To support applications running over TCP, the HoneyTank needs to be able to successfully establish TCP connections. To ensure that the HoneyTank is able to emulate a large number of IP addresses, it is important to design and implement it as a stateless system. Hopefully, for most applications it is possible to infer a suitable response packet by simply looking at the contents of the request packet. In this section, we first describe how to correctly accept TCP connection establishments. We later show how to support standard applications running over TCP.

To establish a TCP connection, a client and a server exchange three TCP segments. The first segment sent by the client has its `SYN` flag set and contains a sequence number chosen by the client (x). The server replies with a segment with the `SYN` and `ACK` flags set and a sequence number (y) chosen by the server. This segment acknowledges the received `SYN` segment by containing $x + 1$ as its acknowledgement number. The client will reply to this `SYN+ACK` with a segment whose acknowledgement number is equal to $y + 1$.

Real TCP implementations create state for the new connection either upon arrival of the first (`SYN`) or third (`ACK`) segment and maintain state during the entire TCP session.

Our HoneyTank establishes a new TCP connection by sending the appropriate `SYN+ACK` segment to each `SYN` segment received. All TCP segments containing only an acknowledgement and no data such as the third segment of the three-way handshake are ignored.

For some protocols, a “welcome banner” must be sent by the server directly after the three-way handshake. For those protocols, in addition to the packet containing the `SYN+ACK`, we create and send immediately to the client a packet containing the banner.

The implementation in RUSSEL is rather straightforward. Two rules are running in parallel. The first rule tests the TCP flags of the received segment which are exported by the format adaptor built on top of `libpcap` [30]. If the record is a `SYN` segment, then we select an initial sequence number and send the corresponding `SYN+ACK` segment in reply with the adequate sequence number.

The second rule tests if the current record is an `ACK` segment and observes the contents of the TCP payload. If it is empty, the received segment is silently discarded. Otherwise, we analyze its content and we send the required reply segment (see later). A new instance of both rules are retriggered for the next received packet.

So far, a human interacting with the HoneyTank could easily detect that the servers are emulated. A simple method to detect the HoneyTank is to send a TCP segment containing data on a non-established TCP connection. A normal TCP implementation would return a `RST` segment, a stateful firewall would drop the segment and the HoneyTank will acknowledge the segment as if the connection was already established. This problem has been solved by relying on the acknowledgement numbers. During the three-way handshake, when the first `SYN` segment arrives, we use a hash function to select the sequence number to put in the `SYN+ACK` segment sent in reply. This

sequence number is $H(IP_{src}.IP_{dst}.port_{src}.port_{dst}.Secret)$ where *Secret* is a configuration parameter of the HoneyTank.

When receiving a TCP segment with the ACK flag set, the hash value $H(IP_{src}.IP_{dst}.port_{src}.port_{dst}.Secret)$ is computed. This hash value is then compared with the acknowledgement number present in the packet. We define as a configuration parameter an “acceptance window” around the hash value. If the ack number falls inside this windows (i.e. is “near” the hash value), this segment is assumed to be part of an initiated TCP session. Otherwise, the segment is silently discarded.

During the three-way handshake, current TCP implementations include TCP options to negotiate parameters of the TCP connection such as the Maximum Segment Size (MSS) [32] or the utilization of TCP extensions such as the window scale option, the timestamps option [33] or the selective acknowledgements. The MSS option indicates the maximum size of the segments to be sent on the TCP connection. Each TCP end-systems indicates the largest segments which it is able to accept on the connection. In practice, most TCP implementations use the minimum value proposed by both end-systems with a minimum of 536 bytes. Our current implementation of HoneyTank assumes this default MSS value but will accept any unfragmented TCP segment.

The most useful TCP extension for the HoneyTank is the TCP timestamp option. Most current TCP implementations support this option [33]. This option was developed to improve the accuracy of the round-trip-time measurement and to protect high bandwidth-delay product TCP connections against wrapped sequence numbers.

Each TCP segment may contain a timestamp option. If the client supports this extension, it will include a null timestamp option in the SYN segment. If the server also supports this option, it will include it in the SYN+ACK segment. A timestamp option contains two 32 bits timestamps : *TSval* and *TSecr*. When a TCP entity sends a TCP segment, it will place in the *TSval* field a 32 bits integer, usually derived from its clock. The *TSecr* will contain the last value received from the other entity in the *TSval* field.

For example, consider an emulated server willing to verify that the client correctly finishes the three-way handshake. Without maintaining state in the HoneyTank, we need to distinguish between the ACK segment finishing the three-way handshake and a normal ACK in the flow of an established connection. With the timestamp option, a simple solution to this problem is to copy the sequence number of the SYN+ACK segment inside the *TSval* field of the timestamp option. When replying to this SYN+ACK segment, the client will echo this value in the *TSecr* field of the ACK segment. By comparing the *TSecr* field and the acknowledgement number, the HoneyTank will easily determine whether it is the third segment of a three-way handshake or a segment sent during the normal data flow. We will provide more details about the TCP timestamp options in the HoneyTank in section IV-D.

C. Emulation of stateless protocols

The first server we implemented in the HoneyTank provides the echo service. This standard service is supported

by most TCP/IP implementations. The client establishes a TCP connection to the server and all the information sent by the client is echoed by the server. To emulate this service, the HoneyTank simply replies to each received data segment with a data segment constructed as follows : the payload of the reply segment is equal to the payload of the received segment. The sequence number of the reply segment is set to the acknowledgement number of the received segment plus one. The acknowledgement number of the reply segment is set to the sequence number of the last byte in the received segment plus one. This implies that we assume that all segments were received correctly and in sequence. If TCP segments are lost, the HoneyTank will not send the acknowledgements, allowing the sender to retransmit the lost segments. As the objective of the HoneyTank is to receive malicious traffic, this is not a severe problem.

Another example of a stateless application-level protocol is the HyperText Transfer Protocol (HTTP) [34]. HTTP is widely used and several worms spread through vulnerabilities in various HTTP servers [35]. A HTTP server accepts TCP connections established by clients on port 80. After the three-way handshake, the client sends a request. This request can be a single command with one parameter, for example `GET index.html` or a one line command with parameters and additional information in subsequent lines. The server analyzes the received request and sends a response composed of a status line, a header with optional information and often a MIME document.

The simplest approach to emulate a HTTP server without maintaining any state is to assume that when the client sends a HTTP request, it will send it inside a single TCP segment. In this case, the server can use regular expressions to match the content of the received TCP segments with the standard HTTP commands and send the corresponding responses. If no matching rule is found for the received TCP segment, the HoneyTank acknowledges the received segment. Table I shows the regular expressions used in our HoneyTank to emulate HTTP. The configuration of an emulated HTTP server can be tailored to fake different types of HTTP servers by providing different `Server` identification strings in the HTTP responses.

Using regular expressions to determine the command sent by the client is not perfect from a theoretical viewpoint. A first possible problem is that the request may be sent inside several TCP segments instead of a single one. In practice, when testing the HoneyTank with several HTTP clients, we found the emulation to be sufficient. A second possible problem is that a MIME document containing HTTP commands may be attached to a HTTP request. This would happen for example when a client is using a POST request to send to the emulated server the RFC defining HTTP. However, in practice this is unlikely to happen. As our objective is to collect malicious traffic, an incorrect reply to a malicious request is not a severe problem.

Although many application-level protocols use ASCII encoded commands and replies, some important applications exchange binary messages. As an example of such protocols, we consider the Remote Procedure Call (RPC) [36] based services

TABLE I
EMULATION OF HTTP/1.1

Regular expression	Reply
GET .* HTTP/1.1\n	HTTP/1.1 200 Ok\nHTML page
HEAD .* HTTP/1.1\n	HTTP/1.1 200 Ok\nLast-modified:...
CONNECT .* HTTP/1.1\n	HTTP/1.1 200 Ok\n\n
TRACE .* HTTP/1.1\n	HTTP/1.1 200 Ok\nContent-type:message/http\n\n...
OPTIONS .* HTTP/1.1\n	HTTP/1.1 200 Ok\nAllow: CONNECT,...\n\n
PUT .* HTTP/1.1\n	HTTP/1.1 201 Created\n\n
DELETE .* HTTP/1.1\n	HTTP/1.1 200 Ok\n\n
POST .* HTTP/1.1\n	HTTP/1.1 200 Ok\n\n

which are often targeted by worms. Among the available RPC services, the portmapper/rpc.bind [37] is important as it listens to a standard TCP port and provides the list of RPC services supported on the local system and the TCP/UDP ports used to reach those services. A request to the portmapper is a standard RPC call addressed to the service number 100000 on port 111. To emulate the portmapper service in our HoneyTank, we created a rule which waits for a RPC message on the default portmapper. This request is encoded in XDR format and contains the service number and a transaction identifier. We extract this identifier and use it to build a matching RPC reply containing a list of fake RPC services which are emulated on the HoneyTank.

D. Emulation of stateful protocols

As another example of protocol implemented on the HoneyTank, we consider the Simple Mail Transfer Protocol (SMTP) [38]. SMTP is the standard protocol used to send electronic mail. SMTP servers are subject to different types of attacks such as buffer overflows or spammers using them as relays to send unsolicited emails. SMTP is a stateful application level protocol running over TCP on port 25.

During a SMTP session, an email is transferred in a few steps. After the SYN+ACK segment of the TCP three-way handshake, the server sends its banner. The client sends a HELO message and the server confirms by sending a one-line reply containing a number and a comment. After this exchange, a mail transaction can start. The client issues a MAIL FROM command indicating the sender's email address. The server replies and indicates whether the sender is acceptable or not. Then, the client issues one or several RCPT TO commands that are confirmed by the server. Finally, the client uses the DATA command to send the entire email followed by a single line containing ".". The arrival of the email is confirmed by the server.

The main problem to emulate a SMTP server is to correctly reply to the mandatory SMTP commands [38]. In our prototype, we implemented a simple solution based on regular expressions. When a TCP segment arrives, we check whether its payload matches one of the regular expressions of table II. If so, a TCP segment containing a positive reply is sent and the received segment is acknowledged. Otherwise, we assume that the received segment is part of the body of an email message being received after a successful DATA command and we simply acknowledge the segment.

We verified that this implementation based on regular expressions was sufficient to interact correctly with several SMTP clients. This accuracy in the SMTP emulation is probably sufficient to track simple spammers trying to exploit vulnerable SMTP servers. But since no state is maintained, a human attacker can easily detect that he is interacting with a fake SMTP server. For example, our server will accept a DATA command even if it is not preceded by a MAIL FROM and RCPT TO command. Similarly, if the string "RCPT TO: \n" appears in the DATA section of a mail our server will send a "250 OK" reply rather than ignore it.

The regular expression approach is sufficient to emulate a simple SMTP server. However, its behaviour is far from a normal server maintaining state. To emulate such a stateful server without maintaining any state in the HoneyTank, we need to determine the state of the emulated server from each TCP segment received from the client.

With the normal TCP protocol we used until now, it is not possible. However, the TCP timestamps can be used to store the state of the SMTP in the client as follows.

We will use the TSval field of the TCP timestamp to store information about the state of the emulated server. When a packet is received by the server, the TSecr field contains the last value of the TSval field which has been received by the client. In order to respect the TCP timestamp specification, the TSval value must be strictly increasing between each packets.

To achieve this, we consider that the first 24 most significant bits of the field are seen as a counter that will be incremented by one each time a packet is sent. We call this the "counter" part of TSval. This counter is incremented for each TCP segment sent. The last 8 least significant bits are used to store the state of the emulated server. We call this the "state" part of TSval.

First, we place the value 1 in the state part of TSval in the TCP segment containing the banner. This timestamp value indicates that the server is expecting the HELO command. Upon arrival of a TCP segment with the state part of TSecr set to 1, the SMTP server will only accept a HELO. The reply message sent in response to this command will contain a state set to 2. This timestamp indicates that the SMTP server is expecting a MAIL FROM command. A state value of 3 is used when the SMTP server expects the first RCPT TO command. After this command (state=4), the client can either send another RCPT TO command or an email by using the DATA command. During the transfer of the email message,

TABLE II
EMULATION OF SMTP WITH REGULAR EXPRESSIONS

Regular expression	Reply
HELO .* \n	250 Requested mail action okay, completed
MAIL FROM:.* \n	250 Requested mail action okay, completed
RCPT TO:.* \n	250 Requested mail action okay, completed
DATA \n. \n	354 Start mail input; end with <CRLF>.<CRLF>
NOOP \n	250 Requested mail action okay, completed
QUIT \n	221 Closing
TURN...	502 Command not implemented

TABLE III
EMULATION OF SMTP WITH TCP TIMESTAMPS

State received	Regular expression	State sent	Reply message
1	HELO .* \n	2	250 Requested mail action okay, completed
1	MAIL RCPT DATA	1	503 Bad sequence of commands
2	MAIL FROM:.* \n	3	250 Requested mail action okay, completed
2	HELO RCPT MAIL DATA	2	503 Bad sequence of commands
3	RCPT TO:.* \n	4	250 Requested mail action okay, completed
3	HELO MAIL RCPT DATA	3	503 Bad sequence of commands
4	RCPT TO:.* \n	4	250 Requested mail action okay, completed
4	DATA \n	5	354 Start mail input; end with <CRLF>.<CRLF>
4	HELO MAIL	4	503 Bad sequence of commands
5	\n. \n	2	250 Requested mail action okay, completed
5	!(\n. \n)	5	<i>no reply, received segment is acknowledged</i>
<5	QUIT \n	999	221 Closing
<5	VRFY EXPN HELP...	TSecr	502 Command not implemented

the server sends TCP segments with the state part of TSval set to 5. In any state besides the email message transfer, the server will correctly reply to a QUIT and an invalid command. Table III shows an emulated SMTP server using TCP timestamps.

The TCP timestamps can also be used to recover from some segment losses and reordering of TCP segments. For example, consider a SMTP client sending the MAIL FROM: command in two segments, the first containing MAIL and the second FROM: a@a.com. If the first segment is lost, the HoneyTank will receive a segment with the state part of TSecr set to 2 that does not match. The received segment should be ignored (and thus not acknowledged) by the HoneyTank. As the TCP implementation on the client maintains state, it will retransmit the two missing segments. Furthermore, as TCP implementations support the Nagle algorithm, the client will retransmit all its unacknowledged data in a single segment.

E. TCP connection clearing

A TCP connection is composed of three phases : the three-way handshake, the data transfer phase and the clearing phase. TCP supports two types of connection clearing : a graceful clearing with the exchange of segments with the FIN flag set and an abrupt clearing with the utilization of the RST flag.

In the current version of HoneyTank, we do not emulate the clearing of a TCP connection. When a TCP segment is received with the RST flag, this segment is ignored. HoneyTank does not send TCP segments with the FIN flag set and we choose to also ignore the TCP segments received with the FIN or RST flag set.

Most worms are implemented as applications that interact with the TCP implementation on the infected operating system. A worm will typically first call the connect system call to verify whether the target responds to TCP connection establishments. The infected operating system maintains TCP state for this established connection. Then, the worm will send its exploit by using usually up to a few send system calls and will either wait for an answer or close the TCP connection by using the close system call. On most TCP implementations, the default behavior of this system call is that the operating system will first try to gracefully clear the TCP connection by sending FIN segments. As the HoneyTank does not reply to those segment, the infected operating system will retransmit the FIN segment several times before abruptly closing the connection and sending a RST segment. During those retransmissions, the current thread of the worm is blocked on the close system call. This could slowdown the propagation of worms.

V. EXPERIMENTAL RESULTS

To evaluate the benefits of using a HoneyTank to collect malicious activities, we attached our prototype HoneyTank to a campus network's router. The campus network is a class B network. This experiment was made on June 25th 2004. The router was configured as described in figure 1. Requests sent to IP addresses belonging to unused subnets were thus routed to the HoneyTank. At the time of the experiment, more than 40000 IPv4 addresses were in unused subnets of the class B campus network.

The HoneyTank was configured to emulate HTTP and

SMTP¹ as shown in tables I and II and to accept TCP connections and acknowledge all received segments on the other ports. Due to the architecture of the campus network, it was unfortunately impossible to place the HoneyTank outside of the firewall while collecting all those addresses. The campus firewall used simple rules to block traffic on classical ports used by worms, trojans, P2P applications and NetBIOS².

Despite the campus firewall, the HoneyTank received a lot of malicious traffic. As an example, we consider the operation of the HoneyTank during 5.5 hours³. During this period, the HoneyTank received 3.433.579 TCP segments from 12859 distinct IP addresses and sent 1.936.767 TCP segments. 1.702.632 received TCP segments contained data. The amount of traffic collected is much more important than expected.

Figure 3 shows ports that are the most targeted in our trace. The well known ports are annotated with the registered name at IANA. We have noticed during our different experiments that the port distribution was rapidly changing over time, but that HTTP port (80) was always in the top 10. The most attacked port in our trace (6129) is attributed to the Dameware Mini Remote Control Protocol for which an exploit exists since December 2003. The second top port in our trace (9898) is probably the Dabber worm which uses a vulnerability in the FTP server component of the Sasser worm. HTTPS arrives at the third place but this service was not emulated by the HoneyTank. The fourth port, 2745, is commonly known as a backdoor port set up by the Beagle worm. The 1025 port is used by the Microsoft Remote Procedure Call (RPC) service and it is the only Microsoft specific port that was not filtered by the campus firewall. Maybe the attacks on the port 19150 were targeted at `gkrellmd` which is a system monitor (CPU, mail, memory, ...).

To compare the HoneyTank with a Darknet [8], we built two different packet traces. The first one is the libpcap trace of all TCP packets sent and received by the HoneyTank. The second is the trace that would have been collected by a Darknet. As a Darknet does not respond to any received packet, we built this trace by extracting from the HoneyTank trace only the received TCP SYN segments (a Darknet could receive non SYN segments but they are rare. During our experiment, 6.5% of the TCP flows did not start with a SYN).

We used Snort on the two libpcap traces with all default rules enabled and the bleeding rules from [39]. In the Honeytank trace (table IV), we see that the most active rule used by Snort is "SHELLCODE x86 NOOP". It is a generic rule which searches for a stream of 14 consecutive NOP instructions (0x90) in the payload of the packets and which indicates potential buffer overflows. We also see a lot of web attacks (prefixed by `http_inspect`) based mostly on classical bugs in Microsoft IIS. The rule called "SCAN Proxy Port

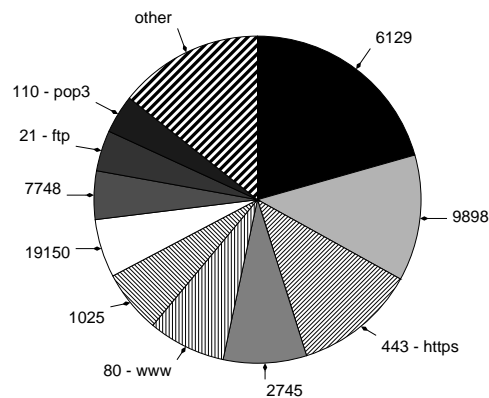


Fig. 3. Destination port distribution of TCP SYN segments

1080 attempt" only checks for a SYN on the tcp port 1080. We do not think that it was really related to SOCKS (1080) as the received data was not a valid SOCKS command. Instead, we think that it is used by trojans such as SubSeven 2.0 which also uses the 1080 port, although we have no proof of it. The "typot trojan" rule is triggered every time a SYN segment is seen with a TCP window size of 55808 bytes and is generated by the Linux typot trojan. Note that for most cases, several machines are responsible for the attacks.

The right column in table IV shows whether the same rule was also triggered by Snort in the Darknet trace. Clearly, Snort detects many more attacks when we let the worm or the attacker go further than a single SYN segment. Obviously, all HTTP attacks and SHELLCODE attacks are only seen in the HoneyTank trace. In the Darknet trace, the HTTP attacks cannot be distinguished from port scans. With this additional information, we can decide with more certainty that the source is malicious.

We also evaluated the flow length distribution on the Honeytank trace. 62% of the 312.464 flows contained less than 10 packets and 25% contained between 10 and 20 packets. In terms of TCP payload, about 50% of the flows contained no data and about two thirds contained less than 10 bytes of payload.

VI. LIMITATIONS OF THE HONEYTANK

In this section, we evaluate the limitations of the HoneyTank and the stateless approach.

The HoneyTank was designed to fool computer worms but it is not intelligent enough to fool a human intruder. A human interacting with the HoneyTank could easily detect that the servers are emulated. A solution to this problem is to emulate servers accurately by maintaining state using the TCP timestamps solution discussed in section IV-D. However, TCP timestamps are optional and if the remote client does not offer the TCP timestamp option in its initial SYN packet the HoneyTank cannot use this option in its replies.

Nevertheless, when interacting with worms, this kind of behavior from the HoneyTank is not a strong limitation.

¹During this experiment, we used the stateless implementation (without TCP Timestamps trick) of the SMTP server. The implementation using TCP Timestamps was used during another experiment which is not described here.

²TCP ports : 12345-12346, 27374, 16959, 27374, 10008, 12378, 1433, 3127-3198, 4751, 5554, 9996, 6346-6347, 1214, 1285, 1299, 1331, 1337, 3135-3137, 4242, 4661-4663, 4665, 4881, 6257,6699, 8875-8876, 6881-6889, 135-139, 445.

³An anonymized version of the collected trace can be obtained by contacting the authors via email.

TABLE IV
SNORT ALERTS ON THE HONEYTANK TRACE

Snort Rule Name	Unique sources	Occurrences	Found in Darknet trace
SHELLCODE x86 NOOP	461	910996	no
(http_inspect) BARE BYTE UNICODE ENCODING	772	10720	no
WEB-MISC WebDAV search access	368	8719	no
(http_inspect) OVERSIZE REQUEST-URI DIRECTORY	376	8701	no
SCAN SOCKS Proxy attempt	99	4378	yes
(http_inspect) DOUBLE DECODING ATTACK	64	2603	no
WEB-IIS cmd.exe access	441	2385	no
WEB-IIS unicode directory traversal attempt	64	2357	no
WEB-IIS ISAPI .ida attempt	401	606	no
(http_inspect) NON-RFC HTTP DELIMITER	406	604	no
(http_inspect) U ENCODING	402	600	no
Portscans	323	435	yes
WEB-IIS CodeRed v2 root.exe access	80	345	no
WEB-MISC http directory traversal	58	312	no
BAD-TRAFFIC tcp port 0 traffic	28	266	yes
BACKDOOR tygot trojan traffic	160	222	yes
SCAN Proxy Port 8080 attempt	86	192	yes
WEB-MISC robots.txt access	5	6	no
SCAN FIN	1	2	no
(http_inspect) OVERSIZE CHUNK ENCODING	1	1	no

Worms often rely on the infected operating system and use the standard socket system calls to infect remote computers. This implies that the worm is usually not able to send raw TCP segments. Most current TCP implementations support the TCP timestamps option. Furthermore, the timestamp options found in the received segments are not exposed by the operating system to the application. Thus, a worm could only rely on the messages received from the distant server to detect whether it is a real server or a HoneyTank. By sending appropriate banners and responses, it is possible to fake different server implementations [17].

As the HoneyTank does not maintain any state, it is not vulnerable to DoS attacks such as a TCP SYN flood. On the contrary, honeypots maintaining state are vulnerable to such attacks. For example, Honeyd [17] maintains state for each TCP connection and launches a process to handle each connection or supports it via a python interpreter. Faced with a SYN flood attack or simply a TCP worm spreading from a large number of machines, a honeyd using processes would saturate the process table of its host operating system. On many Unix variants, up to 32,000 processes can run at the same time. Faced with a SYN flood attack, the HoneyTank would simply send the required SYN+ACK replies. If the load on the access link of the HoneyTank becomes too high, we could limit the amount of reply segments sent by responding only to a subset of the received segments.

A potential problem with the HoneyTank is that it could be used by an attacker as a packet amplifier to flood distant networks. For example, consider an attacker sending spoofed TCP segments containing GET a.html to a HoneyTank emulating a HTTP server as described in table I. The HoneyTank will reply to those segments with TCP segments containing a HTTP header and the beginning of an HTML file. To reduce the impact of this problem, we limit the size of the TCP segments sent by the HoneyTank and try to ensure that the reply sent by the HoneyTank is not much larger than the

received segment. We can also configure the access router to do shaping on the packets sent by the HoneyTank in order to limit the output bandwidth.

As the HoneyTank does not maintain any state, it cannot correctly handle IP fragments. In the current prototype, the received fragments are silently discarded.

Currently, the HoneyTank cannot handle compressed or encrypted connections as they require to maintain some state. By using the TCP timestamp options, it could be possible, at least in theory, to place the value of the key required to decrypt the segment in the TSval field so that each received encrypted segment contains the decryption key. Unfortunately, the timestamp fields are only 32 bits wide and most encryption keys, both public and private, are much larger. However, this limitation does not mean that the HoneyTank could not, at least partially, emulate application-level protocols using encryption to slowdown the spreading of worms.

Finally, the HoneyTank, like many implementations, could suffer from buffer overflow problems. However, it is very unlikely to happen since the HoneyTank only treats TCP segments individually, and the maximum size of each segment is limited by the LAN interface of the HoneyTank. Furthermore, the only operations performed by the HoneyTank are regular expression matches and simple computations. The content of the received segments is never stored in variable-length buffers.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have first discussed several methods which can be used to efficiently collect malicious traffic in operational ISP and campus or enterprise networks. For campus networks, we have shown that Mobile IPv4 can be used to efficiently collect information about local scans. Once deployed, a collector should be able to collect and analyze both UDP and TCP-based malicious activities.

Then we have proposed two methods to emulate a large number of TCP-based services without maintaining any state in the workstation receiving the malicious traffic. Our first method replies to all TCP SYN segments to accept all TCP connection establishment attempts and matches the payload of the received data segments to determine a plausible reply. Our second method relies on the TCP timestamp option supported by most TCP implementations and uses it to “store” the state of the emulated server in the distant client. This provides a more accurate emulation of the TCP services.

We have implemented a prototype HoneyTank in the ASAX IDS. We have used it to emulate TCP services on about 40,000 IPv4 addresses. During a short period of time, it was able to capture many types of malicious activities.

The approach described in this paper could be expanded in several ways. Firstly, new TCP-based services should be added to the prototype. Most clear-text application-level protocols can be easily supported. We believe that using the high-level RUSSEL language can ease the implementation of new protocol responders. Secondly, the prototype should be deployed in various locations, including on existing network telescopes or on Darknets operated by large ISPs to collect and analyze malicious traffic during large periods of time. Thirdly, a large deployment of HoneyTanks could slowdown the spreading of worms as they would try to infect many non-existing machines.

Finally, it would be useful to compare the performances between our stateless implementation and a stateful implementation. This comparison should measure both the scalability of the implementations and the quality of the malicious traffic captured.

REFERENCES

- [1] APWG, “Anti phishing working group,” <http://www.antiphishing.org/>.
- [2] Darrell M. Kienzle and Matthew C. Elder, “Recent worms: a survey and trends,” in *Proceedings of the 2003 ACM workshop on Rapid Malcode* [40], pp. 1–10.
- [3] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham, “A taxonomy of computer worms,” in *Proceedings of the 2003 ACM workshop on Rapid Malcode* [40], pp. 11–18.
- [4] Marty Roesch, “Snort,” <http://www.snort.org>.
- [5] CERT, “CERT Advisory CA-2001-26 Nimda Worm,” <http://www.cert.org/advisories/ca-2001-26.html>.
- [6] CERT, “Code Red II : Another Worm exploiting buffer overflow in IIS indexing service DLLAdvisory CA-2001-26 Nimda Worm,” http://www.cert.org/incident_notes/in-2001-09.html.
- [7] Mark Fullmer, “flow-dscan : Detect scanning and other suspicious network activity,” <http://www.splintered.net/sw/flow-tools/>.
- [8] The Team Cymru, “The team cymru darknet project,” <http://www.cymru.com/Darknet/index.html>, June 2004.
- [9] V. Yegneswaran, P. Barford, and D. Plonka, “On the design and use of internet sinks for network abuse monitoring,” in *RAID 2004 Symposium*, September 2004.
- [10] IPMON, “Packet trace analysis,” <http://ipmon.sprint.com/packstat/packetoverview.php>, 2004.
- [11] Cisco, “NetFlow services and applications,” White paper, available from http://www.cisco.com/warp/public/7_32/netflow, 1999.
- [12] E. Gauthier, “Life on a university network: An architecture for automatically detecting, isolating, and cleaning infected hosts,” NANOG30, <http://www.nanog.org/mtg-0402/gauthier.html>, February 2004.
- [13] Vinod Yegneswaran, Paul Barford, and Johannes Ullrich, “Internet intrusions: global characteristics and prevalence,” in *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 2003, pp. 138–147, ACM Press.
- [14] DShield-team, “Distributed intrusion detection system,” <http://www.dshield.org/>.
- [15] David Moore (CAIDA), “Network telescopes: Observing small or distant security events,” http://www.caida.org/outreach/presentations/2002/usenix_sec/, August 2002.
- [16] David Moore, Geoffrey M. Voelker, and Stefan Savage, “Inferring internet Denial-of-Service activity,” in *Proceedings of the 2001 USENIX Security Symposium*, 2001, pp. 9–22.
- [17] Niels Provos, “Honeyd,” <http://www.citi.umich.edu/u/provos/honeyd/>.
- [18] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek, “The click modular router,” ACM Transactions on Computer Systems, August 2000.
- [19] The HoneyNet Project, *Know Your Enemy: Revealing the Security Tools, Tactics, and Motives of the Blackhat Community*, The HoneyNet Project, 2002.
- [20] Marc Dacier, Fabien Pouget, and Hervé Debar, “Honeypots: Practical means to validate malicious fault assumptions,” in *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC’04)*. 2004, pp. 383–388, IEEE Computer Society.
- [21] M. Dacier, F. Pouget, and H. Debar, “Attack processes found on the internet,” in *NATO Symposium IST-041/RSY-013*, Toulouse, April 2004.
- [22] Vincent Berk, George Bakos, and Robert Morris, “Designing a framework for active worm detection on global networks,” in *Proceedings of the First IEEE International Workshop on Information Assurance (IWIA’03)*. 2003, p. 13, IEEE Computer Society.
- [23] C. Kreibich and J. Crowcroft, “Honeycomb - Creating Intrusion Detection Signatures Using Honeypots,” 2nd Workshop on Hot Topics in Networks (HotNets-II), 2003.
- [24] G. Huston, “Bgp table report,” <http://bgp.potaroo.net>, 2004.
- [25] C. Perkins, “Rfc 3344 : IP Mobility Support for IPv4,” <http://www.ietf.org/rfc/rfc3344.txt>, August 2002.
- [26] P. Biondi, “Scapy,” <http://www.cartel-securite.fr/pbiondi/projects/scapy.html>.
- [27] N. Habra, B. Le Charlier, A. Mounji, and I. Mathieu, “ASAX: Software Architecture and Rule-based language for Universal audit trail analysis,” in *Proceedings of the third European Symposium on Research in Security (ESORICS’92)*, Toulouse, Nov. 1992, Lecture Notes in Computer Science, Springer-Verlag.
- [28] A. Mounji, B. Le Charlier, N. Habra, and D. Zampuniéris, “Distributed Audit Trail Analysis,” in *Proceedings of the Internet Society Symposium on Network and Distributed System Security (ISOC’95)*, San Diego, California, Feb. 1995, IEEE.
- [29] A. Mounji, *Languages and Tools for Rule-Based Distributed Intrusion Detection*, Ph.D. thesis, Institute of Computer Science, University of Namur, Belgium, September 1997.
- [30] Tcpcdump-team, ,” <http://www.tcpcdump.org/>.
- [31] Mike D. Schiffman, “The libnet packet construction library,” <http://www.packetfactory.net/Projects/Libnet/>.
- [32] J. Postel, “Transmission control protocol,” Request for Comments 793, Internet Engineering Task Force, Sept. 1981.
- [33] V. Jacobson, R. Braden, and D. Borman, “TCP extensions for high performance,” Request for Comments 1323, Internet Engineering Task Force, May 1992.
- [34] R. Fielding, UC Irvine, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Rfc 2616 : Hypertext Transfer Protocol – HTTP/1.1,” <http://www.ietf.org/rfc/rfc2616.txt>, August 1992.
- [35] CERT, “CERT Advisory CA-2001-19 ”Code Red” Worm Exploiting Buffer Overflow in IIS Indexing Service DLL,” <http://www.cert.org/advisories/CA-2001-19.html>.
- [36] R. Srinivasan, “RPC: remote procedure call protocol specification version 2,” Request for Comments 1831, Internet Engineering Task Force, Aug. 1995.
- [37] R. Srinivasan, “Binding protocols for ONC RPC version 2,” Request for Comments 1833, Internet Engineering Task Force, Aug. 1995.
- [38] J. Postel, “Simple mail transfer protocol,” Request for Comments 821, Internet Engineering Task Force, Aug. 1982.
- [39] “Bleeding snort ruleset,” <http://www.bleedingsnort.com>.
- [40] ACM, *Proceedings of the 2003 ACM workshop on Rapid Malcode*, Washington, DC, USA, Oct. 2003. ACM Press.