CTCP: A Transparent Centralized TCP/IP Architecture for Network Security

Fu-Hau Hsu Tzi-cker Chiueh

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
Email: {fhsu, chiueh}@cs.sunysb.edu

Abstract

Many network security problems can be solved in a centralized TCP (CTCP) architecture, in which an organization's edge router transparently proxies every TCP connection between an internal host and an external host on the Internet. This paper describes the design, implementation, and evaluation of a CTCP router prototype that is built on the Linux kernel. By redirecting all packets targeting at non-existent or non-open-to-public ports to a CTCP socket which pretends to be the original receivers, CTCP could confirm the real identification of the packet sources, collect suspicious traffic from them, and make an illusion that the scanned target ports are all open, thus renders port scanning an useless effort. Under CTCP architecture, external hosts only interacts with a secure CTCP router; therefore, any OS fingerprinting attempt and DoS/DDoS attack targeting at TCP/IP implementation bugs could be thwarted. Moreover, By further checking traffic originating from confirmed scanners, the CTCP router can actually identify buffer overflow attack traffic. Finally, the CTCP router solves the TCP connection hijacking problem by introducing an additional check on the sequence number filed of incoming packets. Despite providing a rich variety of protection, the CTCP architecture does not incur much overhead. On a 1.1GHz Pentium-3 machine with gigabit Ethernet interfaces, the throughput of the CTCP router is 420.3 Mbits/sec, whereas the throughput of a generic Linux router on the same hardware is only 409.1 Mbits/sec.

1. Introduction

Traditionally the TCP/IP stack[1] is implemented at each end host. This paper advocates a centralized TCP/IP (CTCP) architecture in which an organization's edge router transparently splits each TCP connection between an internal host and an external host into two TCP connections, one between the internal host and itself, and the other between itself and the remote host. As a result, the only TCP/IP stack that remote hosts get to interact with is the one on the edge router (called CTCP router hereafter) and many network security problems can be easily solved in this architecture.

Many denial of service attacks [39] exploit implementation bugs in the TCP/IP stack or weaknesses in the TCP/IP specification. For example, the Ping of Death attack creates an IP packet that exceeds the maximum IP packet size allowed according to the IP protocol specification (65,536 bytes), and sends it to the victims, which may crash, hang, or reboot when they receive such a packet. The Teardrop attack sends a series of IP fragments with overlapped offset fields to the victims, which may crash, hang, or reboot when trying to reassemble them. Weaknesses in the TCP/IP specification leave it open to SYN flood attacks. The Smurf attack is a brute-force attack targeted at a feature in the IP specification known as direct broadcast addressing. Under the CTCP architecture, as long as the TCP/IP stack of an organization's CTCP router is correctly implemented, none of the above DOS attacks are possible with its internal hosts. In other words, CTCP greatly simplifies the process of "patching" TCP/IP stack implementation.

Many network attacks start with OS fingerprinting and port scanning to first identify the OS type and the set of services in a remote host, and then determine the proper attack strategy. Both OS fingerprinting and port scanning are typically based on reactions of a sequence of probe packets. Therefore, if these probe packets can be detected, it is possible to identify not only the attacking hosts but also the packets used in the attack. Under the CTCP architecture, it is impossible to fingerprint the OS type of internal hosts because it is the CTCP router that responds to probe packets. Moreover, the CTCP router is in a better position to detect port scanning activities because it examines the

[1] We use the term TCP/IP stack to refer to the entire Internet protocol suite, including UDP and ICMP.

source/destination address/port number fields in TCP packets. Finally, the CTCP router provides an effective platform for applying honey-pot technique [15] to capture attack packets.

A network intrusion detection system (NIDS) compares packets against a signature database to identify potential attack packets. There are well known techniques [21, 22] to invade the detection of such systems. Basically these invasion methods exploit differences in interpreting certain parts of an incoming packet between the TCP/IP stack on an NIDS and that on an end host, for example, the TTL field and overlapped IP fragments. Under the CTCP architecture, none of these invasion techniques work because there is only one TCP/IP stack to the outside world, and the NIDS is based on its interpretation of the incoming packets.

In addition to security benefits, the CTCP architecture also offers several performance advantages. First, because all TCP connections with external hosts terminate at the CTCP router, it can manage their congestion control windows by taking into account the bandwidth sharing effect among connections that go to the same remote subnet [10]. One immediate benefit is that a new TCP connection's congestion window does not have to grow from scratch, but from some larger value that past history suggests is appropriate. Second, because the CTCP router performs *connection splicing* for every TCP flow, it provides an additional level of indirection that is useful in such applications as server load balancing and fault tolerance.

This paper presents the design, implementation, and evaluation of a fully operational CTCP router that is built on the Linux kernel. To demonstrate the usefulness of this CTCP router, we show how it can be used to prevent OS fingerprinting, to detect port scanning, to identify buffer overflow attack packets, and to stop TCP connection hijacking attacks. In addition, the throughput of the CTCP router prototype is actually slightly higher than that of a generic Linux router on the same hardware. This shows that it is feasible to implement the CTCP architecture on a gigabit/sec router, which should be more than enough for most enterprises' connection to the Internet.

This paper is organized as follows. Section 2 describes the specific security threats that the CTCP router prototype addresses in this paper. Section 3 details the system architecture of the CTCP router, and the functions of its components. Section 4 presents the results of testing the effectiveness and performance of the CTCP router prototype. Section 5 reviews previous efforts to deal with the security issues that the CTCP architecture addresses. Section 6 concludes this paper with a summary of the major contributions, and a brief outline of on-going work.

2. Motivations

In this section, we discuss the main motivations behind the CTCP architecture, in particular, prevention of OS fingerprinting and port scanning, centralized implementation of TCP/IP stack, and buffer overflow attack detection.

2.1. Reconnaissance Deterrence

A typical network attack proceeds in the following stages. First, the attacker scans the Internet to determine the operating system and the services on each host that responds to the scan packets. Then, the attacker attempts to compromise a remote host based on vulnerabilities known to exist on the host's associated OS/services combination. If any of the attacks succeeds, the attacker then installs the attack program on the victim host to include it in the future attack. Because the process of "recruiting" new attack hosts is completely automated, this worm-like network attack can increase the number of attack hosts exponentially and eventually covers most of the vulnerable hosts on Internet within a few minutes to an hour [26].

TCP and UDP ports are a host's communication channels with other hosts. A port is called *open* when there is a program listening on it. Through an open port a TCP/UDP-based application program could exchange data with other software. To identify the set of open ports on a remote host, a port scanning program can simply connect to all possible ports, and determine the set of ports on which the host actually listens. According to the TCP/IP protocol, an open port behaves differently than a close port. For example an open port must reply a SYN packet with a SYN/ACK packet and a close TCP port must reply a SYN packet with a RST packet. Hence, by observing a remote host's responses to carefully crafted packets, a port scanner can deduce whether a port is open on the host. Based on this principle, a wide variety of port scanning tools [1, 2] have been developed.

The TCP/IP protocol specification leaves some room for implementation flexibility, such as the initial sequence number, the initial window size, the DF bit, the ToS setting, how two fragments with overlapped offsets should be handled, etc. The TCP/IP stack implementations of different operating systems can freely choose how to exploit this flexibility. As a result, it is possible for an OS fingerprinting tool to identify an operating system based on its responses to a sequence of probe packets. The collective response of an operating system to this packet sequence constitutes its *fingerprint*. For example, when a TCP/IP stack receives a packet that is neither SYN nor ACK and is destined to a non-open port, the correct response according to RFC 793 is not to respond; however, many implementations such as MS Windows, BSDI, CISCO, HP/UX send back a RESET packet. Some OS fingerprinting tools are active in that they

send probe packets to target machines to collect information. Nmap [3] and Xprob [4] are most popular active fingerprinting tools. Others are passive in that they just observe the traffic associated with the monitored host to collect information. The value of the Time to Live (TTL) field, the initial window size in the TCP header, and the values of DF bit and TOS field in the IP header are the common monitor targets. p0f [5] and siphon [6] fall into this category.

One way to defeat the automated attack scheme used by worms is to stop them at the reconnaissance phase, i.e., preventing them from knowing the OS type and network services on the innocent hosts. The centralized CTCP architecture effectively hides the OS type information of internal hosts because OS fingerprinting tools can only see the TCP/IP stack of an organization's CTCP router.

2.2. Detection of Buffer Overflow Attack

Buffer overflow attack [11, 12] is one of the most significant security threats to the Internet today. It overwrites some control-sensitive data structure (a return address or function pointer) of a victim application so that the application's control is re-directed to an injected code or a libc function [13] (*return-to-libc* attack). Various approaches have been proposed to solve this problem. However, most if not all of them involve modifications to the applications or operating systems on the end hosts. As a result, their adoption in practice has been rather limited.

The key to a successful buffer overflow attack is to successfully overwrite the target control-sensitive data structure in the victim application. However, the exact address of the target control-sensitive data structure may vary from instance to instance even for the same source code for the following reasons:

- Environment variables and command line arguments (including the name of the command) will influence the location of the main() function's stack frame, hence the stack frames of all subsequently called functions.
- Due to the alignment requirement, a compiler doesn't necessarily allocate memory for variables according to the order they appear in the source code.
- For the same code, different compilers used by different OSes could create different memory layout for the same set of variables. In other words, for a C program, the memory layout of a set of variables created by a Linux host could be different from the one created by a Solaris host.
- Address obfuscation [14] compilers insert byte strings into memory areas for variables to further change the memory layout. The length of the inserted byte string is randomly generated at compile time or at run time.

Hence, to maximize the success rate of a buffer overflow attack, attackers typically repeat the string used to overwrite the target control-sensitive data structure multiple times in the attack packets. In the case of return-into-libc attacks, the repeating string consists of the entry point of the libc function, the previous frame pointer, and the input arguments. In other cases, the repeating string consists of the entry point of the injected code only. For the 6 exploit strings [8, 9] we checked (LFTP, ATPHTTPd, in.telnetd, samba, INND, and TCPdump) the repeating times are all above 10.

2.3. Stopping TCP Connection Hijacking

Sequence number implicitly plays an authentication role in TCP connections. When a TCP packet whose sequence number is outside the associated socket's receiving window is received, it is dropped and an ACK packet that includes the expected receiving window information is sent back. On the other hand, if an incoming TCP packet is accepted by a socket, then the socket's receiving window changes accordingly. Therefore, if an attacker can correctly guess the sequence number of an on-going TCP connection, she can both send forged data to and change the receiving window of an end point of this connection. For a TCP connection between two hosts, H_a and H_b, sending forged packets to H_a could cause it to ignore data sent by H_b; moreover, if H_a sends any reply to the forged packets, both the hijacker and H_b will receive it. The above two results will create abnormal and detectable behavior at H_b whose most likely response is to close its socket. The close will not close H_a's socket immediately, because H_b's packets will be dropped by H_a. After H_b's close, any packet of the hijacked connection from H_a will result in H_b's sending back a RST packet which has correct sequence number and will close H_a's socket.

However, if the hijacker changes both hosts' receiving windows at the same time and temporarily suppresses their responses to the forged packets before the hijacking is finished, as Joncheray's [34] DO-NOT ECHO case does, then she can take over the TCP connection silently and each of the original two hosts is fooled into thinking it is communicating with the other host as usual when in fact it is communicating with the attacker. In this case, packets sent by one host are always dropped by the other host, and additional ACK packets are generated, which in turn are dropped and trigger more ACK packets. This positive feedback loop creates a *TCP ACK storm*. To defeat the TCP connection hijacking , special attention should be paid to the above silent hijacking. With the CTCP architecture, once the TCP/IP stack on the CTCP router incorporates such special processing, all the internal hosts are protected from the TCP connection hijacking.

3. CTCP System Architecture

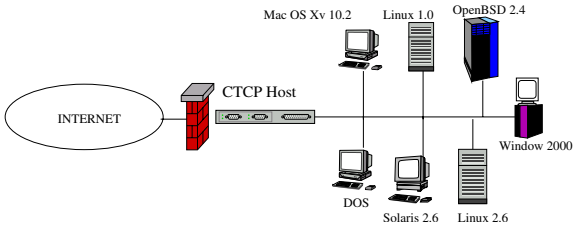

Figure 1. *Geographic Location of A CTCP Router.*

A CTCP router performs the same role as a standard edge router for an organization, and oversees all the packets coming in and getting out of that organization, as shown in Figure 1. As a result, the TCP/IP stack on the CTCP router is the stack with which all remote hosts interact. In the following, we will call TCP/IP packet header fields (e.g. protocol type field, TTL, FLAGS, and IP identification field), their length and the retransmission time-out values (RTOs) [7], as *Transmission Meta Data* (TMD).

3.1. Strategies

In the CTCP architecture, the CTCP router splits each TCP connection between an inner host and a remote host into two TCP connections using a special listening socket called *gate socket* and the nonlocal binding mechanism [27, 28]. This split-connection structure allows the CTCP router to relay only the payload portion of incoming/outgoing packets, but none of their TMD. Consequently, malicious TMD from attackers can never reach inner hosts and TMD from inner hosts that can potentially reveal their OS type never get to remote attackers.

In addition, the CTCP router redirects all packets targeting at non-existing hosts or non-open-to-public ports to a CTCP socket called *police socket*, which is created by an user-level process called the *operative*. The operative performs two functions. First, it creates an illusion that there is a host behind every public IP address and every port on each of such hosts is open, essentially rendering port scanning a useless exercise. Second, it tries to interact with remote attackers that attempt to access non-existent hosts or non-open ports so that it can collect their IP address and the attack packets. Once attack hosts are identified, TCP traffic from these hosts is also redirected to the police socket.

Traffic collected by the operative is in turn given to another user-level process called MCI (malicious code incubator), which uses a heuristic method to identify buffer overflow attack and return-into-libc attack. This heuristic is based on the following observation: most if not all exploit strings used in buffer overflow attacks and return-into-libc attacks include repeated patterns in order to increase the likelihood of overwriting a certain control-sensitive data structure in the victim program. For a buffer overflow attack, the repeated pattern is the entry point of a piece of injected code, which is most likely on the stack. For a return-into-libc attack, the repeated pattern includes the entry point of the target libc function, the address of its first argument, and its arguments. To further restrict the values in the repeated pattern, we exploit the fact that entry point addresses of libc functions must be aligned on a 4-byte boundary, and that they must be within a certain range of the address space. For example, in Linux, the user-level stack starts from address 0xbfffffff and grows downward [16]. The default maximum size of a process's user-level stack is 2Mbytes [38], but because the average function frame size is 28 bytes [31, 32], most programs are not supposed to use a 2Mbyte stack. In our test, a 8k stack is enough to detect the 6 exploit code. A Linux shared library should be within the range that starts at address 0x40000000 and ends at the beginning of the stack, i.e., 0xbfffffff-2M.

Based on the above facts, MCI uses the following rule to recognize buffer overflow attacks: If an input string contains a stack address that repeats 3 times, then it is regarded as a buffer overflow attack; if an input string contains at least 3 copies of a special libc pattern mentioned above, then it is regarded as a return-into-libc exploit string. An input string here refers to all the bytes sent from an outside host to an inner host in a TCP connection.

When a host, say H_a, receives a TCP packet whose sequence number is outside its receiving window and whose source is H_b, there are two possibilities. First, this is a packet sent by the other end of a hijacked TCP connection, i.e. H_b. Second, this is a packet that somehow gets trapped in the Internet and later re-emerges. A trapped packet's ack number should be smaller than or equal to H_a's sending window. However if the packet belongs to a hijacked connection, it should contain an ack number larger than H_a's sending window, because in order to avoid H_b's receiving H_a's responses to the hijacker's packets, the hijacker must increase H_b's receiving window before H_a sends any payload packet. Thus, whenever CTCP receives a packet whose sequence number is outside the receiving socket's receiving window, CTCP further checks the packet's ack number. If the ack number is larger than the local socket's sending window, then CTCP sends back a RST packet with the rejected packet's ack number as its sequence number. This way, CTCP avoids disconnecting a connection due to the arrival of trapped packets but disconnect those connections that are hijacked.

3.2. System Components

A CTCP router consists of 4 major components and two listening sockets as shown in Figure 2. The *Traffic Arbitra-*

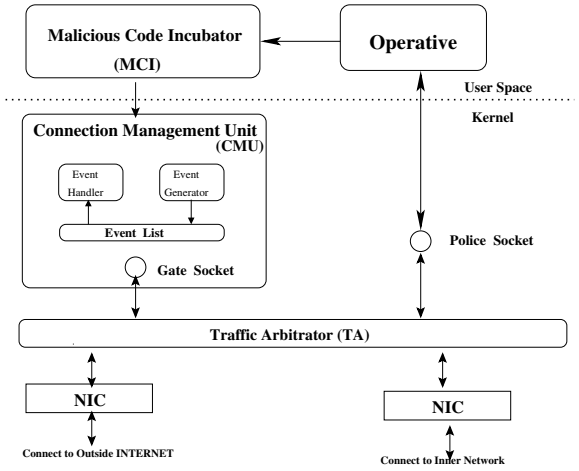

Figure 2. *System Structure of A CTCP Router. Each Block Represents A System Component. An Arrow Represents Data Flows between Components.*

tor (TA) and the *Connection Management Unit* (CMU) are in the kernel. The other two components, the *operative* and *Malicious Code Incubator* (MCI), are user-level processes. The *police socket* is created by the operative. The operative and MCI only process suspicious traffic. So, for normal traffic processing there is no context switch between user-level code and kernel-level code. In addition, for normal traffic there is no data copying between the user and kernel address space.

TA's major responsibility is to assign incoming packets to appropriate handlers which could be the Connection Management Unit (CMU), the operative, or itself. Because CTCP receives and transmits data on behalf of the inner hosts, it breaks each TCP connection into two separate sub-connections: One sub-connection links a CTCP port to an inner host port and the other joins an outside host port and another CTCP port. One of the above two TCP ports is always a clone of the listening gate socket. Each one of the above two sub-connections is called the *buddy connection* of the other. CTCP uses CMU to manage these sub-connections and pipeline data between a sub-connection and its buddy connection(see Figure 3).

Connection Management Unit (CMU) consists of two major components, event generator and event handler, and two major data structures, event list and data bridge. The event generator is responsible for transforming packets coming from TA into corresponding events and then appending them into the event list along with the addresses of the packets' receiving sockets. An event can be the finish of a three way handshaking, data arrival, or a disconnection request. The event handler takes events from the event list and processes the events according to their con-

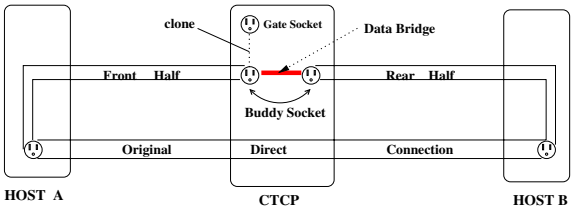

Figure 3. *Connection Division. Each TCP Connection Is Split into Two Sub-connections at CTCP.*

tents. The data bridge is shared by a sub-connection and its buddy connection, and is used to exchange data (not packets) between them. When detecting stealthy TCP hijacking indicator packets, CMU is responsible for disconnecting the hijacked connection through RST packets.

TA redirects suspicious TCP traffic, which includes all packets destined to non-existent hosts and non-open-to-public ports, to the operative. After gathering enough packets from potential attackers, the operative gives them to the Malicious Code Incubator (MCI). These two components are designed to confirm the identifications of attacker hosts and check whether the collected packets contain buffer overflow attack exploit strings or return-into-libc exploit strings using the heuristics described above.

3.3. Data Flows Inside a CTCP Router

When a packet arrives at CTCP, it is given to the traffic arbitrator first. According to its addresses, TA assigns it to an appropriate handler, which could be CMU, the operative, or the TA itself if it decides to drop the packet. The 3 main data flow paths within the CTCP router are:

1. $NIC_o \Leftrightarrow$ TA $\Leftrightarrow$ CMU $\Leftrightarrow NIC_i$
2. $NIC_o \Leftrightarrow$ TA $\Leftrightarrow$ Operative $\Leftrightarrow$ MCI $\Leftrightarrow NIC_h$
3. $NIC_o \Leftrightarrow$ TA

Packets on legitimate connections, e.g., incoming connections to open-to-public ports or outgoing connections, travel through the first path. Suspicious packets take the second path. And all unsafe and unnecessary packets are dropped through the third path. Finally, under the following situations TA drops packets directly without handing them to any handler, because either these packets could reveal critical security-related information of inner hosts or these packets contain attack code:

- ICMP and UDP packets heading to inner hosts.
- Incoming packets match an attack signature.

4. Effectiveness Analysis and Performance Evaluation

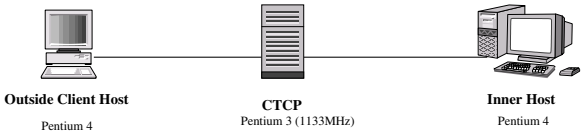

Figure 4. *Experiment Setup. All hosts use Gigabit NICs.*

This section presents experiment results about the effectiveness and throughput of CTCP. Figure 4 describes the test setup. Outside untrusted clients are executed at a Pentium 4 machine. CTCP resides at a Pentium 3 (1133MHz) host. Inner servers are run at a Pentium 4 computer. All 3 machines are equipped with Intel Pro/1000 Giga bit NICs and run Linux 2.4.7.

4.1. Effectiveness Analysis

In this subsection we analyze the effectiveness of CTCP in defending against the security threats we are addressing in this paper.

4.1.1 OS Fingerprinting

In OS fingerprinting test, we use both the active fingerprinting tool, Nmap, and the passive tool, p0f, to probe the inner host's OS type with or without CTCP. Table 1 shows the test results.

Nmap detects an inner host's OS type by actively sending probe packets to its targets. However, open ports and closes ones need different probe packets. E.g. the result of sending a FIN probe packet to an open port could tell a scanner whether the inner host's OS type is among MS Windows, BSDI, CISCO, HP/UX, MVS, and IRIX or not. But the result of sending a FIN packet to an close port can not tell the scanner so much information. Hence, in order to infer the scanned target's OS type, Nmap must find at least one open port and one close port at the target so that it can decide which probe packets should be sent to which ports. Therefore the first phase of OS fingerprinting is port-scanning. During this phase plenty of port-scanning packets, e.g. SYN packets, are sent to different ports of the target.

In the first test of Nmap, the inner host is disconnected. When the middle host is a Linux router, because Nmap can not find any open port at the target, it can not make any deduction about the target's OS type. When the middle host is a CTCP router, because of the work of the operative, all target host's ports look like open ports even though the host is disconnected. But all the port-scanning packets are redirected to the police socket whose backlog queue size is only five, therefore some port-scanning packets, e.g. SYN packets, will be dropped and the scanner can not receive any responses associated with those ports that the dropped packets were heading for. There is no ACK for RST packets, in other words there is no RST packet retransmission, thus Nmap will think those ports are closed and will send them probe packets suitable for close ports. However those ports are open. This confuses Nmap. So even though Nmap is supposed to be able to get the OS type of the host that it interacts with (here the host is CTCP) and misunderstand CTCP's OS type as its target's OS type, Nmap still needs to guess to get the OS type. Fortunately, this time Nmap's guess is right. But even so, this will not cause any problem, because it is the inner hosts that provide network services the scanners are interested in and CTCP doesn't provide those kinds of services. Moreover, CTCP is supposed to have the most secure system installed.

	Linux Router	CTCP Router
Off-Line	X	Linux 2.4.x - 2.5.x(86%)
Windows XP	95/98/ME NT/2K/XP	Linux 2.4.x - 2.5.x(86%)
Linux 2.4.7-10	Linux 2.4.x - 2.5.x	Linux 2.4.x - 2.5.x(86%)

Table 1. *OS types of inner hosts reported by Nmap. The router is either a Linux router or a CTCP router. Both routers run Linux 2.4.7-10 kernel. Each column represents a different router. Each row describes a distinct OS used by the inner host. Within a table entry, 'X' means "unable to detect and* 86% *means the confidence level Nmap has about a guess.*

In the second test of Nmap the inner host executes Window XP. If the middle host is a Linux router, the Nmap can correctly induce inner host's OS type. If the middle machine is CTCP, then for the same reason as above, Nmap still needs to guess. This time Nmap is wrong.

In the third test of Nmap, we install the same OS as CTCP's at the inner host to check whether this can help Nmap avoid guessing. In this test both CTCP and the inner host use Linux 2.4.7-10 OS. Again when CTCP works as a normal router, Nmap can correctly infer the inner host's OS type and version. But when CTCP's protection functions are activated, the results are the same as the second test.

The above results show no matter what OS an inner host uses, Nmap can not get the correct OS type of the inner host. The best Nmap can get is the guessed OS type of CTCP.

The test results of p0h are similar to Nmap's. p0h still misunderstands CTCP's OS type as its target's OS type. But

Stack Size	R=3	R=10
2MB	0/49	0/15
8KB	0/0	0/0

Table 2. *Number of false positives when running the buffer overflow attack detection heuristic against the test samples. "R=3" means the number of repeated patterns in the input string has to be equal to or more than 3. Two stack sizes are used, 2MB And 8KB. In each entry the left is the number of false positives for return-into-libc attack, and the right is the number of false positives for buffer overflow attacks using injected code.*

p0h can accurately deduce, not guess, the OS type of the host that it interacts with (here it is CTCP).

4.1.2 Exploit String Detector

To test the effectiveness of MCI's exploit string detector, object files (library files, executable files, ...), document files (pdf, ps, doc, txt, html, ...), and picture files (gif, jpg, mpeg, ...) with size 209MB, 183MB, and 11MB respectively are used. They are randomly chosen from different hosts. For false negative test, MCI could detect the 6 exploit strings described in subsection 2.2 correctly.

For false positive test, MCI uses signatures introduced in subsection 3.1 to examine the test samples with different stack sizes and distinct numbers of repeating pattern copies. Table 2 shows the results.

The false positive test results show when stack size is 8k and the number of repeating pattern copies is 3, MCI has 0 false positive in examining these 404-Mbyte test samples. Qualitative analysis could explain this results. First, a Linux stack address starts with 0xbf which is not a visible ASCII character, therefore we can anticipate this character will not appear at telnet data, e-mails which don't have attachments, html files and so on. Second even though this character could appear in an executable file or in an image file, in order to cause a false positive alarm, the same pattern must repeat 3 times. For a binary file, it means if there is an instruction with that string pattern, then the exact same instruction must repeat several times in the program. From our experience, it seems it is not a common situation. Based on the above analysis, we think even though it is still possible that the signatures we used will cause false positives, we can expected it will not be high.

The above tests use static test samples (i.e. files in disks). Now we are working on getting the test samples dynamically from the network.

4.1.3 Stealthy TCP Connection Hijacking

Hunt [36] and JUGGERNAUT [35] are two popular TCP hijacking tools. But either the hijacking is visible to the victims (Hunt) or the hijacking tool automatically quits when executed in our system (JUGGERNAUT).

In a TELNET case, when hunt is used, one end host of the hijacked connection can see the letters typed by the hijacker from its own monitor and all this end host's output are dropped by its communication counterpart, moreover when the host terminates the process, the hijacked connection is also closed. This test shows visible TCP hijacking could be detected and terminated by the hijacked connection's owner. Thus it is not an ideal hijacking tool.

We use a packet construction tool, gspoof [37], and TCPDUMP to test CTCP's effectiveness in defending against stealthy TCP connection hijacking.

First a TCP connection is created between two host, HOST-a and HOST-b. Then TCPDUMP is executed at HOST-a. After gspoof is used to construct a packet that has HOST-a as it destination and HOST-b as its source and has a sequence number outside HOST-a's receiving window, the packet is send to HOST-a from a third host HOST-c. The results of TCPDUMP show after receiving the crafted packet, the correct RST packet is created and sent back.

4.2. Performance Evaluation

In performance test, we analyze the throughput impact introduced by CTCP. In the tests, different numbers of TCP connections are opened between the client and the inner server. Then the client continues pumping data into the inner host. And we measure how many bits per second are received by the server. Results in Figure 5 shows that CTCP has a better throughput than a Linux router does. In addition, as a Linux router, CTCP's throughputs are not influenced by the number of connections traversing through it.

Analysis and further experiments show the throughput gains come mainly from two sources. First, under CTCP architecture, a client host has shorter ACK packet returning time, and thus could send packets more quickly than it does under a Linux router. Second, CTCP architecture allows the processing of a packet and the transmission of next packet to proceed simultaneously. In other words, when the client checks the validity of an ACK packet of a previous outgoing packet, say P_1, and prepares the next outgoing packet, say P_2, CTCP could process P_1 simultaneously.

5. Related Work

Proxy server is a widely used structure to secure LANs and improve performance. Under this structure, two communication parties, the client and the server, exchange data

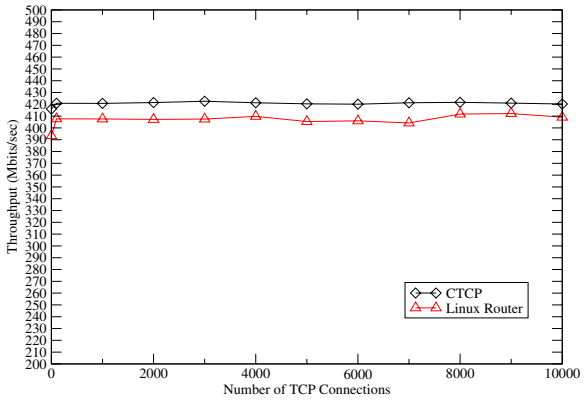

Figure 5. *Throughputs of CTCP and a Linux Router, when the number of TCP connections increases from 1 to 10000.*

through a proxy server. In other words, there is no direct connection between these two entities. The proxy server accesses data on behalf of them and relay data to their counterparts. Under proxy server structures, external hosts only interact with the proxy servers; therefore, they provide good protection against OS fingerprinting.

According to the class of data a proxy server relays, proxy servers could be further divided into two categories, the application-level proxies and circuit-level proxies. TIS [23] and transparent proxies [24] belong to the former. SOCKS [17], Dante [20] and TCPProxy [29] are circuit-level proxies.

Application-level proxies relay protocol-level data between end users of TCP connections; thus, this method is able to provide user-level authentication protection. However in order to forward protocol-level data, a proxy must understand the protocol involved. In other words, each protocol needs its own application-level proxy. Both TIS and transparent proxies are application-level proxies. For transparent proxies, the connection between a client and the proxy is established through redirection. So the proxy is only transparent to the client, not to the server. In TIS's case, the proxy server is not transparent to both hosts; hence, the ID of the proxy is visible to external hosts.

Instead of application-level data, a circuit-level proxy relays TCP-level data between its users; thus, it is a generic proxy. As a new network service appears, without modifications, a circuit-level proxy can process the new application's traffic immediately. SOCKS, Dante, and TCPProxy are all circuit-level proxies. One of the major differences between them lays on the level of modification that need to make on the networking libraries, such as socket-related functions.

CTCP utilizes proxy server as a basic platform to develop different security solutions. However in order to ensure CTCP's effectiveness and efficiency, CTCP removes not only the above problems specific to each proxy server but also the following common problems.

Most of current proxy servers are user-level processes which introduce non-trivial overhead upon systems duo to data copies between user space and kernel space, context switches between different processes and context switches between user code and kernel code.

Besides, currently most proxies are not transparent to their users; hence, one end of a TCP connection may regard a transparent proxy as its communication counterpart. This property may distract other services. First it exposes the proxy's identification, and thus put it under the direct fire of malicious users. Second, applications based on IP addresses, such as bandwidth management tools and trusted hosts, can no long function as expected, because instead of the real sources' addresses, they only see the proxy's address. Third, a vicious user inside a proxied network could easily disable the whole network's access to outside servers whose firwalls filter out packets based on their source IP addresses. All the person need to do is to use the proxy to connect the target and make the target's firewall record the proxy as a bad guy.

Smart et al. [21] solves OS fingerprinting by normalizing the traffic. In other words, their method, fingerprint scrubber, eliminates the personal styles (e.g. initial sequence number, window size ... and so on) and ambiguities from traffic to block OS fingerprinting.

Utilizing similar method, traffic normalizer [22] synchronizes TCP connection's state at both a NIDS and a protected host, and thus thwarts malicious users' attempts to bypass NIDS's detection. The above method introduces only a little overhead over the system and could efficiently block the powerful OS fingerprinting tools — NMAP.

But in order to provide more security, there are still some issues needed to be solved. First, modification of TTL field could disable its functionality and result in invalid packets' circuiting around in the network. Second, normalization could not normalize all exploration traffic. For example, in order to protect itself from SYN flood, some OS stops establishing connections after being unable to finish several, say 8, 3-way handshaking processes. As a result, a scanner can just send 8 forged SYN packets and then makes a real connection to the target system. All these exploring packets will not be blocked by the normalizer; hence, later on if the connection can not be established, then the scanner can deduce the target system's OS type, otherwise, the scanner can know what OS she/he can rule out.

Moreover, both the above methods are not compatible with SYN cookies which is an useful tool to defeat SYN flood. Instead of storing state information of an open connection request (such as initialize sequence number and source IP , destination IP, ... and so on) at a protected

host, SYN cookies encode these information in the initial sequence numbers of the replying SYN/ACK packets. The SYN/ACK packets could quickly consume up the normalizer's memory, because all these SYN/ACK packets come from the trusted side of the normalizer , and the normalizer is asked to establish state information for packets coming from the trusted side. Due to the above reasons, under SYN flood attack, maybe the protected host is still alive because of the protection provided by SYN cookies, but the normalizer is already disable due to running out of memory.

As using repeating addresses in attack strings to increase the chance to have a successful attack, repeating NOP instructions right before the injected code are also widely used in attack strings. Toth and Kruegel's solution [30] focus on detecting the appearance of a sequence of NOP instructions which they call sledge or their equivalences. In their method they try to dissemble the packet content, and if a substring of a packet content could be interpreted as a sequence of 30 or more instructions, an alarm is issued. However it is not a trivial work to disassemble a packet to find the longest execution path which may start at any byte inside that packet. If an attacker on purpose crafts packets that contains numerous execution paths and all of the paths have length less than 30, than the attacker could issue some kind of DoS/DDoS attacks upon this approach without being detected.

As CTCP, buttercup [25] also uses addresses as hints to detect buffer overflow attacks. Instead of using a generic address pattern for all buffer overflow attacks, for each individual attack string, they need to study the targeted program and its buggy overflowed functions to drive a small range of possible address that could be used to launch a successful attack. Later on this address range is used as a signature of the specific attack string; therefore, if any word of a packet's payload could be interpreted as an address within this range, the packet is classified as an attack packet. This method simplifies the signatures of known attacks; thus, improves the performance of signature matching. However, for unknown buffer overflow attacks, current buttercup version doesn't take them into account.

6. Conclusion and Future Work

In this paper we advocate a centralized TCP/IP architecture, in which the only TCP/IP stack visible to the outside world is the one on an organization's CTCP router. To demonstrate its usefulness, we show that it can effectively stop OS fingerprinting and port scanning, the scouting activity for most automated attacks, including worms. In addition, the CTCP architecture greatly facilitate enterprise-scale deployment of solutions to vulnerabilities due to TCP/IP implementation bugs, for example, DoS attacks and TCP connection hijacking attacks. Finally, the CTCP architecture provides a flexible platform for developing honeypots that can apply content filtering techniques to identify and extract attack packets, for example buffer overflow attacks. A major concern about the CTCP architecture is that the additional processing may exert serious performance cost on a CTCP router compared with a generic IP router. Performance measurements on a working CTCP router shows that the performance cost of the CTCP architecture is relatively modest. In fact, the throughput of a Linux-based CTCP router can actually be higher than that of a Linux-based IP router, because the former forces the CPU to pay more attention to Linux's network subsystem.

Our current algorithm for detecting buffer overflow attacks is based on the assumption that some special patterns (e.g. return address) will repeat in the attack payload. However, a patient attacker could bypass this detection method by using attack payload that contains no more than 2 copies of a special pattern. In the future we plan to remove this assumption by applying run-time binary disassembly to suspicious packets. The honeypot in the current CTCP prototype cannot carry out a conversation with remote attackers beyond the initial three-way hand-shake. We plan to improve the honeypot so that it can interact with attackers in the same way that the service it simulates does, all without actually running the service daemon.

References

[1] Fyodor, "The Art of Port Scanning," 1997.

[2] Ofir Arkin, "Network Scanning Techniques," http://www.publicom.co.il, 1999

[3] Fyodor, "Remote OS Detection via TCP/IP Stack FingerPrinting," http://www.insecure.org/nmap/nmap-fingerprinting-article.html

[4] Orfin Arkin, " ICMP usage in scanning," http://www.sys-security.com/html/projects/icmp.html.

[5] P0f README, "P0f README," http://www.stearns.org/p0f/README.

[6] Jose Nazario, "Passive Fingerprinting using Network Client Applications, Nov. 2000, http://www.crimelabs.net/docs/passive.html

[7] Franck Veysset, Olivier Courtay, Olivier Heen, "New Tool And Technique For Remote Operating System Fingerprinting," Intranode Software Technologies, April, 2002.

[8] Fyodor, "Exploit world! Master Index for ALL Exploits, " http://www.insecure.org/sploits_all.html

[9] http://www.securiteam.com/exploits/archive.html

[10] Prashant Pradhan, Tzi-cker Chiueh, Anindya Neogi, "Aggregate TCP Congestion Control Using Multiple Network Probing," ICDCS 2000.

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in Proceedings of 7th USENIX Security Conference, San Antonio, Texas, Jan. 1998

[12] Tzi-cker Chiueh and Fu-Hau Hsu, "RAD: A Compiler Time Solution to Buffer Overflow Attacks," Proceeding of ICDCS 2001, Arizon USA, April 2001

[13] Nergal, "The Advanced Return-into-Lib(c) Exploits," Volume 10, Issue 58, Phrack.

[14] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar, "Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits," 12th USENIX Security Symposium, Washington, DC, August 2003.

[15] Honeynet Porject team, " Honeynet Project," http://www.honeynet.org.

[16] Daniel P. Bovet, Marco Cesati, "Understanding the Linux Kernel, 2nd edition," O'Reilly , Dec 2002.

[17] Permeo, "SOCKS Overview," http://www.socks.permeo.com/AboutSOCKS/SOCKSOverview.asp.

[18] Mark Grennan, "Firewall and Proxy Server HOWTO," http://www.tldp.org/HOWTO/Firewall-HOWTO.html#toc11

[19] Gopinath K. "N. Kernel Support for Building Network Firewalls Based on the Paradigm of Selective Inspection of Packets at Applicatin Level," Thesis, Indian Institute of Technology, Kanpur April 1999.

[20] Inferno Nettverk, "Dante," http://www.inet.no/dante/.

[21] Matthew Smart, G. Robert Malan, Farnam Jahanian, "Defeating TCP/IP Stack Fingerprinting," USENIX Security Symposium, Aug. 2000.

[22] Mark Handley, Vern Paxson, and Christian Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," Proc. USENIX Security Symposium 2001.

[23] Trusted Information Systems, "TIS Firewall Toolkit," http://www.tis.com

[24] Daniel Kiracofe, "Transparent Proxy with Linux and Squid Mini-HOWTO," http://en.tldp.org/HOWTO/ Transparent-Proxy.html.

[25] A. Pasupulati, J. Coit, K. Levitt, S.F. Wu, S.H. Li, R.C. Kuo, and K.P. Fan, "Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities," Network Operations and Management Symposium 2004(NOMS 2004).

[26] Stuart Staniford, Vern Paxson, Nicholas Weaver, " How to Own the Internet in Your Spare Time," Proceedings of the 11th USENIX Security Symposium, 2002.

[27] Nadav Har'El, "Bug in Nonlocal-bind (Transparent Proxy) ?," http://www.cs.helsinki.fi/linux/linux-kernel/2001-22/0678.html.

[28] Alexey Kuznetsov,"Re:Bug in Nonlocal-bind (Transparent Proxy) ?," http://search.luky.org/linux-kernel.2001/msg32060.html.

[29] Wolfgang Zekoll, "tcpproxy - Generic TCP/IP Proxy," http://www.quietsche-entchen.de/software/tcpproxy.html.

[30] Thomas Toth, Christopher Kruegel, "Accurate Buffer Overflow Detection via Abstract Payload Execution," Distributed Systems Group, Technical University Vienna, Austria, RAID 2002.

[31] D. Ditzel and R. McLellan., "Register Allocation for Free: The C Machine Stack Cache," Proc. of the Symp. on Architectural Support for Programming Languages and Operating Systems, pp. 48 - 56, March 1982.

[32] Sangyeun Cho, Pen-Chung Yew, Gyungho Lee, "Decoupling local variable accesses in a wide-issue superscalar processor," Pro. of the 26th annual international symposium on Computer architecture, Georgia, United States, 1999.

[33] CERT, "CERT Advisory CA-2001-09 Statistical Weaknesses in TCP/IP Initial Sequence Numbers," http://www.cert.org/advisories/CA-2001-09.html

[34] Laurent Joncheray, "Simple Active Attack Against TCP," 5th USENIX UNIX Security Symposium, June 1995.

[35] route—daemon9, "JUGGERNAUT," Volume 7, Issue 50, Phrack maganize.

[36] Krauz's, Pavel. "HUNT Project." 1.5 - bug fix release. 30th May 2000. URL: http://lin.fsid.cvut.cz/ kra/index.html(9th February, 2001)

[37] Embyte, "gspoof," http://gspoof.sourceforge.net/

[38] Sandeep Grover, "Buffer Overflow Attacks and Their Countermeasures," Linux Journal, March 10, 2003.

[39] Deokjo Jeon, " Understanding DDoS Attacks, Tools and Free Anti- tools with Recommendation," http://www.sans.org/infosecFAQ/threats /Understanding_ddos.htm.