Scalable Network-based Buffer Overflow Attack Detection

Fu-Hau Hsu
Department of Computer Science and Information Engineering
National Central University
Taoyuan, Taiwan, R.O.C.
hsufh@csie.ncu.edu.tw

Fanglu Guo
Symantec Research Laboratory
Cupertino, CA, U.S.A.
fanglu_guo@symantec.com

Tzi-cker Chiueh
Computer Science Department
Stony Brook University
Stony Brook, NY, U.S.A.
chiueh@cs.sunysb.edu

ABSTRACT

Buffer overflow attack is the main attack method that most if not all existing malicious worms use to propagate themselves from machine to machine. Although a great deal of research has been invested in defense mechanisms against buffer overflow attack, most of them require modifications to the network applications and/or the platforms that host them. Being an extension work of CTCP, this paper presents a *network-based* low performance overhead buffer overflow attack detection system called *Nebula*[1], which can detect both known and zero-day buffer overflow attacks based solely on the packets observed *without requiring any modifications to the end hosts*. Moreover, instead of deriving a specific signature for each individual buffer overflow attack instance, *Nebula* uses a generalized signature that can capture all known variants of buffer overflow attacks while reducing the number of false positives to a negligible level. In addition, *Nebula* is built on a centralized TCP/IP architecture that effectively defeats all existing NIDS evasion techniques. Finally, *Nebula* incorporates a payload type identification mechanism that reduces further the false positive rate and scales the proposed buffer overflow attack detection scheme to gigabit network links.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection (e.g., firewalls)*

General Terms

Security

Keywords

Buffer Overflow Attacks, Return-into-libc Attacks, CTCP, Generalized Attack Signatures, Payload Bypassing, Network-based Intrusion Detection

[1] NEtwork-based BUffer overfLow Attack detection

ANCS'06, December 3–5, 2006, San Jose, California, USA.

1. INTRODUCTION

Buffer overflow attack is arguably the most widely used and thus most dangerous attack method used today. It accounts for more than 50% of all the security vulnerabilities recorded by CERT [20]. Many solutions to the buffer overflow attack problem have been proposed in the last decade, including compiler transformation approaches that detect and/or prevent tampering of control-sensitive data structures [2, 3, 19], library rewriting approaches that ensure each incoming packet never steps beyond the corresponding packet-receiving buffer's bound, and operating system approaches that prevent malicious code injected by buffer overflow attacks from being executed. In theory, these efforts have largely solved the buffer overflow attack problem. In practice, however, new buffer overflow vulnerabilities are still discovered and reported on a routine basis. This discrepancy between theory and practice arises because almost all existing solutions to the buffer overflow attack problem require substantial modification to the computing infrastructure in which network applications are developed or executed, and thus have met substantial resistance in actual deployment. One way to overcome this deployment problem is to develop a network-based buffer overflow attack detection mechanism that can detect arbitrary buffer overflow attacks without requiring any changes to the network applications or the hosts they run on. This paper describes the design, implementation and evaluation of such a system, called *Nebula*.

Existing network-based intrusion detection systems (NIDS) compare incoming packets against an attack signature database, and raise an alert when one or multiple matches are found. Typically, a separate signature is created for each distinct buffer overflow attack. Obviously, this approach cannot effectively detect zero-day attacks, whose signature is unavailable by definition, or variants of known attacks. Moreover, under this approach, false positives are inevitable and tend to be numerous, mainly because the signature matching logic in NIDSs rarely takes into account the context in which buffer overflow attacks take place. In contrast, the design goal of *Nebula* is to detect *arbitrary* buffer overflow attacks, zero-day or not, based solely on the payload of incoming packets. While *Nebula* is still signature-based, the signature it uses is designed to capture all known buffer overflow attacks. Although the *Nebula* prototype described in this paper does not achieve its design goal completely, we believe it represents an important step toward reaching that goal.

There are two variants of buffer overflow attack: *code-injection* (CI) attack, where attackers insert a piece of malicious code into the victim application's address space and then steer the application's control to the injected code; *return to libc* (RTL) attack, where attackers directly steer the control of the victim application to a function pre-existing in its address space, e.g., a library function. In both cases, attackers hijack the control of the vic-

tim application, by modifying a control-sensitive data structure such as a return address and changing it to either a location on the stack (CI attack) or a location in the text or code region (RTL attack). From the above analysis, a buffer overflow attack packet must include a 4-byte *hijack destination word* that corresponds to a memory address on the stack or in the text region. Furthermore, to increase the success probability and robustness of a buffer overflow attack, attackers almost always replicate the hijack destination word in the packet so as to accommodate differences in the address of the target control-sensitive data structure due to different combinations of compiler, loader, operating system, and command-line arguments. In summary, the main signature that *Nebula* uses to detect buffer overflow attacks is a sequence of identical 4-byte words that correspond to an address in the stack region or text region.

For all the buffer overflow attacks whose actual attack packets are publicly available, the above signature can detect them all without any false negatives. However, the above signature could also trigger many false positives, especially when the hijack destination word is assumed to appear in the attack packet only once or twice. *Nebula* addresses the false positive problem using a multi-pronged approach. First of all, *Nebula* can recognize files downloaded via FTP, HTTP, P2P file sharing, and BitTorrent-like applications, and exclude bytes in downloaded files from the signature matching process. This optimization significantly improves the run-time efficiency and decreases the false positive rate of *Nebula*. Secondly, *Nebula* can detect and record failed buffer overflow attacks, which eventually lead to the victim application's termination, and use this information to detect an attacker's attempt to map out the victim application's exact address space layout by repeated probing. Finally, *Nebula* transparently learns the normal connection establishment/tear-down behaviors of internal servers, and uses this information to detect any connectivity anomaly after an attacker successfully compromises an internal server.

The rest of this paper is organized as follows. Section 2 reviews previous works on host-based and network-based buffer overflow attack detection. Section 3 describes the design of *Nebula*, including its multi-pronged approach to buffer overflow attack detection. Section 4 presents a detailed evaluation of *Nebula* in terms of its false negatives, false positives, and run-time overhead. Section 5 concludes this paper with a summary of major research contributions and an outline for future work.

2. RELATED WORK

Traditionally, detecting buffer overflow attacks at the network packet level is deemed as a difficult work because packets provide too little information about the attacked programs. However with the effort of security community, several promising methods have been proposed.

As using repeating addresses in attack strings to increase the chance to have a successful attack, repeated NOP instructions right before the injected code are also widely used in attack strings. Toth and Kruegel's solution [14] 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.

Like *Nebula*, buttercup [13] 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 specific vulnerable 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, it seems, current buttercup version doesn't take them into account.

Andersson et al.'s method [15] uses system calls as a buffer overflow attack signature; therefore, if more than two threat level one system calls [16] are detected, then an alert is issued. Currently their algorithm assume system calls are issued in the following form. mov system_call_num, %eax; int 80h. However there are other ways to issue a system call, such as push system_call_num; pop %eax; int 80h.

The basic processing unit of all the above methods is a packet. In other words, the content of one packet will not influence the examination result of other packet. As a result, if an attacker could split her/his input into several small packets, not fragments, then she/he can bypass the detection; moreover, because in return-into-libc buffer overflow attack strings, there is no injected code, the two approaches which search for buffer overflow attack strings based on binary code can no longer handle this problem. For buttercup approach, currently they haven't handled the return-into-libc problem yet.

Kruegel et al. [22] and Ke Wang et al. [23] (PAYL) base on the ASCII code frequency distribution in the payloads of network traffic to/from a network service to build a normal traffic payload model for that service. By comparing the actual traffic into/from a network service with its associated model at run time, these systems can detect abnormal payload. To achieve low false positive rate, PAYL computes a traffic payload model for each traffic direction and for each distinct port and service.

PAYL assumes that it is possible to build a reliable traffic payload model for any given network service. It is not clear whether this assumption holds for services whose traffic payloads change frequently, such as a news web site like CNN, a auction web site like Ebay, an E-commerce web site like Amazon, etc. Moreover, like all anomaly detection, PAYL also suffers from the usual "emulation" attach in which the attacker tries to faithfully follow the traffic payload model when composing the attack payload.

3. DESIGN

3.1 Principles of Buffer Overflow Attacks

In a typical buffer overflow attack, the attacker injects an instruction sequence into the victim application *and* transfers the control of the application to the injected code. As an application's text segment is typically read-only, the only way to hijack the control of an application is to dynamically modify the target address of its branch instructions whose target is not fixed at compile time. Such dynamic branch instructions include function returns, pointer-based function calls, and C-style switch statements. These branch instructions typically have their target addresses stored in some stack or heap variable. If an attacker can overflow an array or buffer that resides beside a target-address variable in an application, she can then modify this target address, and eventually take control of the application after the dynamic branch instruction using the target-address variable is executed.

In all known buffer overflow attacks, the predominant way to hijack a victim application's control is by altering the return address of a function that contains an "overflowable" buffer; hence, when the function returns, code stored in the address pointed by the overwriting address will be executed. As an example, the following shows a code segment that is vulnerable to buffer overflow attacks and its stack layout at the time when it is called. SP and FP are the stack pointer and base (or frame) pointer, respectively. The stack grows downward toward address zero.

```
main() {                          STACK LAYOUT
    Input();
}                                 Return Address of Input()
                                  Base Pointer of main() ;FP
                                  Local Variable i
Input() {                         UserID[4]
   int i;                         UserID[3]
   int UserID[5];                 UserID[2]
                                  UserID[1]
   i=0;                           UserID[0]              ;SP
   while ((scanf("%d",
    &(UserID[i]))) != EOF)
        i++;
}
```

In this case, the procedure `Input()` keeps accepting inputs into the array `UserID` without stopping even after the bound of `UserID` is exceeded. As a result, the attacker can fill the stack frame of `Input()` with whatever values she desires, including the stack location holding the return address of `Input()`. As a result, when `Input()` returns, the control is transfered to wherever the overwritten return address points to. Typically the attacker sets the new return address to the beginning of a piece of injected code, for example, the address of `UserID[0]`. The ability to transfer the control of a victim application to a sequence of instructions that is under the control of the attacker opens the door to an infinite number of further compromises. However, this flexibility is only possible if the injected code can execute from the stack or heap.

A variant of buffer overflow attack that does not require injecting any code is called *return-to-libc* attack, which transfers the control of the victim program to the entry point of a prechosen system call function, e.g., exec(). As a result, the attacker can still inflict upon the underlying system without injecting any code. However, it is difficult to set up a *return-to-libc* attack because the arguments of the target system call must be carefully laid out on the stack in order for the desired attack to take effect. The following figure shows the stack frame for the function above, which contains a buffer overflow vulnerability, before and after a *return-to-libc* attack.

```
10044                                10008
10040                                X
10036 Return Address of Input()      Entry to exec()
10032 Base Pointer of main(); FP     X  ;new SP
10028 Local Variable i               X
10024 UserID[4]                      X
10020 UserID[3]                      X
10016 UserID[2]                      X
10012 UserID[1]                      "/csh"
10008 UserID[0] ; SP                 "/sys"
```

By overflowing the array `UserID[]`, the attacker is able to modify the return address, and inserts a pointer (10008) at the memory location 10044 as the input argument to the target system call `exec()`. More specifically, the attacker changes the return address of `Input()` to the entry point of `exec()`. Before the return instruction in the function `Input()` is executed, the stack pointer register (SP) is assigned to the value in the frame pointer register (FP) and thus points to 10032. Then the top of stack is popped and copied to FP. At this point, SP points to 10036. After the return instruction is executed, SP points to 10040 and the program's control goes to the entry point of `exec()`. Inside `exec()`, the content of FP is pushed to the top of stack and SP points to 10036, and the value in SP is then copied to FP. By convention, the first input argument resides in the memory location [FP] + 8, or 10044 in this case. The content of memory location 10044 points to a 8-byte character string ("/sys/csh"). As a result, the attacker is able to make the system call `exec("/sys/csh")`.

3.2 Generalized Signature

The key challenge to a successful buffer overflow attack is how to overwrite the target control-sensitive data structure in the victim application. However, the distance between the "overflowable" buffer (`UserID[]` in the above example) and the target control-sensitive data structure (the return address of `Input()`) may vary for different instances of the same network application for the following reasons:

- Due to memory alignment requirement, different compilers may allocate local and global variables in an order that is completely different from the order they appear in the source code.
- For the same source code, different compilers on different OSes may use a different memory layout for the same set of variables. In other words, the memory layout of a C program's variables on by a Linux host could be different from that on a Solaris host.
- Address obfuscation compilers [4] insert byte strings into memory areas for variables to further randomize the memory layout. The length of the inserted byte string is randomly generated at compile time or at run time.

Because the distance between the overflowable buffer and the target control-sensitive data structure is not completely predictable, buffer overflow attack authors typically repeat multiple times the attack string used to overwrite the target control-sensitive data structure, so as to maximize the success rate. In the case of RTL attacks, the attack string that is repeated consists of (1) the entry point of the target libc or system call function and (2) one or multiple pointers to character strings as input arguments. In the case of CI attacks, the attack that is repeated is the entry point of the injected code. For the ten buffer overflow attack exploit strings [12] we examined, the number of times at which the attack string is repeated ranges from 4 to 100 times.

To turn the above observation into a practical signature, further restrictions need to be imposed on the contents of the attack strings. First, in the X86 architecture, memory addresses, including entry points to a piece of injected code or a libc function, must fall within a well-defined address space range. In the case of RTL attacks, the return address entry must point to the address space region in which the libc library resides; moreover, because the character strings that are used as input arguments are on the stack, the input argument pointers must thus point to the stack region. Similarly, for a CI attack, the return address entry must point to the stack region because the injected code typically sits on the stack.

In Linux, the user-level stack starts from address $0xbfffffff$ and grows downward, and the default maximum size of a process's user-level stack is 2 MBytes [8]. However, because the average function frame size is 28 bytes [6, 7], most processes have an active stack size of less than 8 KBytes. This means that the address space range from $0xbfffffff$ to $0xbfffffff - 8K$ should cover all stack variables in most cases. Linux's shared library, which contains all system call entry points, reside in the range that starts at address $0x40000000$ and ends at the beginning of the stack, i.e., $0xbfffffff - 2M$.

To summarize, the generalized signature for recognizing any buffer overflow attacks is as follows: If an input string contains a stack address that repeats N times, then it is regarded as a CI attack; if an input string contains at least N copies of a pattern that consists of a shared library function's entry point address followed by at least one stack address, then it is regarded as a RTL attack. Currently the repeat count N is set to 3.

3.3 Centralized TCP Architecture

For signature matching purpose, *Nebula* considers as a single input string all the bytes that are transferred from an outside host to an inner host over a TCP connection. To collect these input strings, *Nebula* is built on a centralized TCP/IP (CTCP) or TCP proxy architecture [5] in which an organization's edge router (called CTCP router hereafter) transparently splits each TCP connection between an internal host and an external host

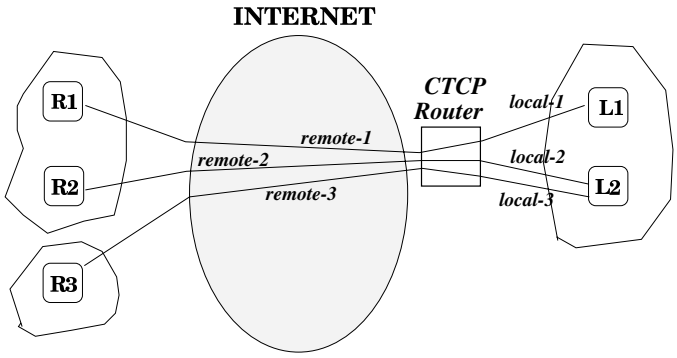

Figure 1: *In the centralized TCP architecture, a network connection an internal host (e.g. L1) and an external host (e.g. R1) is split into two sub- one between the external host and the CTCP router, and the other between the CTCP router and the internal host. As far as R1, R2, and R3 are concerned, it's the TCP/IP protocol stack on the CTCP router that they are interacting with directly. The TCP/IP protocol stacks on L1 and L2 are completely hidden and thus immune from attacks.*

into two TCP connections, one between the internal host and itself, and the other between itself and the remote host, as shown in Figure 1. Packets exchanged over these two TCP connections only contain the original two communication parties' IP addresses; that is, the CTCP router is completely invisible to them. Because the CTCP router is involved in the set-up and teardown of every TCP connection, it can easily identify all the bytes flowing in a TCP connection and presents the resulting stream to *Nebula*'s signature-based filter.

There are well known techniques [17, 18] that can effectively evade the detection of NIDSs. Fundamentally these evasion methods exploit differences in the interpretation of 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 fundamentally there is only one TCP/IP stack visible to the outside world, and the proposed signature-based filtering scheme is based on the CTCP router's interpretation of the incoming packets.

In addition, because the protocol processing associated with all outgoing TCP packets are performed on the CTCP router, it is now possible to collectively manage the congestion-related states of these TCP connections based on their destination subnets [1], and utilize the available bandwidth on the Internet more effectively. In particular, this congestion state sharing mechanism allows short-lived TCP connections to start with a larger initial congestion window size and thus reduces their end-to-end connection time. Similarly, the CTCP router can also serve as the basis for packet reordering detection that could eliminate spurious congestion window reduction.

3.4 Contextual Analysis

The proposed generalized signature covers all known variants of buffer overflow attacks. So it has zero false negative, Moreover, the fact that its signatures are derived from the first principles of buffer overflow attacks, rather than any specific byte patterns from particular buffer overflow attacks, means that *Nebula* can even detect zero-day buffer overflow attacks. However, there is no guarantee that *Nebula* will never generate false positives, especially when the number of repetition patterns N is small and when the stack size is large. We propose *contextual analysis* techniques that analyze bytes surrounding the bytes matching the proposed signature to determine whether these matched bytes indeed correspond to a buffer overflow attack.

From our initial experiments, it is clear that when the number of repeated patterns is assumed to be 3 or more, the number of false positives is negligible. This means that the only way for an attacker to evade *Nebula*'s detection mechanism is to repeat an attack pattern only once or twice. However, because the attacker can never be sure of the exact binary image of the victim program (even with the knowledge of the application's source code), repeating an attack pattern only once or twice is more likely to crash the victim application rather than to take control of it. On both Linux and Windows, when a program is crashed, the OS will terminate all the program's pending socket connections by sending out an RST packet to the communicating hosts on its behalf. Based on this observation, *Nebula* uses the following contextual analysis technique to determine whether an input string containing only one or two copies of an attack pattern indeed represents an attack: checking if a TCP socket connection terminates immediately after receiving an input string that contains one or two copies of a potential attack pattern. When this happens, the packet in question is flagged as a buffer overflow attack packet, and the remote host is flagged as a suspicious host. All future packets from a suspicious host will be examined more thoroughly and critically. This technique effectively discourages attackers from performing trial-and-error experiments with victim applications using an attack packet sequence that repeats the attack pattern only once or twice.

After hijacking a victim application, the attacker typically sets up a separate network connection to either provide a command shell or to download back-door programs. To bypass firewalls, this connection is typically initiated by the victim machine, or is initiated by the attacker machine but aimed at port 80 on the victim machine, which most firewalls allow. From the above analysis, *Nebula* arrives at another contextual analysis technique: after detecting a suspicious input string that contains one or two copies of the proposed signature, *Nebula* checks if the victim machine initiates a new network connection to the attacker machine or the attacker machine initiates a new network connection that targets at port 80 on the victim machine.

3.5 Payload Bypassing

Payload bypassing tries to avoid packet analysis for as much traffic as possible. Because most buffer overflow attacks take place during the exchange of control messages, it is safe to ignore the bulk of data that is downloaded as uninterpreted bytes. For example, in an FTP session, data transferred over the data connection can never be used to mount a buffer overflow attack against the FTP program because the FTP program does not interpret them.

Based on the above observation, we propose to apply an existing network protocol analyzer called Ethereal [9] to identify network packets that correspond to files transferred through HTTP, FTP, and P2P applications such as BitTorrent and eDonkey, and ignore them in the signature matching process. From CacheLogic's measurement [21] on USA, Europe, and Asia backbone in June 2004, HTTP and P2P packets accounted for more than 70% of the total traffic. In P2P traffic, BitTorrent and eDonkey accounted for 70-90%. Thus payload bypassing is expected to significantly cut down the performance overhead associated with *Nebula* . To be sure, this optimization may not be applicable to other content filtering applications, such as those aimed at identifying potentially harmful email attachments.

In addition to the above improvement, payload bypassing also decreases the number of false positives. The leading byte of any words that contain a stack address corresponds to a non-printable ASCII character. Thus, packets exchanged via text-based protocols can never contain bytes that correspond to stack addresses. That is, *Nebula*'s false positives mainly come from packets in binary-based protocols or binary files that are being transferred. Therefore, ignoring downloaded files during signature matching prevents these binary bytes from becoming false positives.

3.5.1 HTTP

Both the control message and payload part of an HTTP transaction are transferred over a single TCP connection. The control message is the message header and the payload is the message body. Most if not all known remote HTTP attacks use the message header. The message body corresponds to uninterpreted byte stream, and contains files requested by the user. To implement payload bypassing for HTTP traffic, *Nebula* only needs to search the `Content-Length` field in the message header. Since the message header normally is only hundreds of bytes at most, this searching overhead is relatively modest. Suppose the length of the following message body is x, then the message body will be the x-byte data after the separator between the message header and message body, and the next message header will start after the X-byte message body.

3.5.2 FTP

Unlike HTTP, an FTP session's control messages travel over a control connection and its files are transferred over a separate data connection. The TCP port number of the data connection is dynamically exchanged in the control connection. Each data connection only transfers one file. To implement payload bypassing for FTP traffic, *Nebula* needs to identify FTP data connections and ignores the data in them. There are two ways for an FTP client and server to exchange the port number of a data connection. The first way is active FTP, in which an FTP client uses the "PORT" command to tell the FTP server the port it will open up and the FTP server connects to the client via that port. The server port for the data connection is typically smaller by one than the control connection's server port number. The second way is passive FTP. In this mode, the FTP server uses the passive response message to tell the client the server port, and the client may use an arbitrary client port number to connect to the FTP server. Thus in passive FTP, the client port cannot be extracted from the control connection. Based on the above analysis, the solution is to match subsequent network connections against the known 3-tuple: server IP, server port, and client IP, and declare the first connection matching this 3-tuple to be the passive FTP data connection. Because FTP's control connection traffic is less than one hundred bytes for each file transfer transaction, the performance cost of searching through the control connection for PORT commands and passive response messages is relatively modest.

3.5.3 BitTorrent

There are two types of BitTorrent traffic. The first type is query traffic from BitTorrent nodes to tracker nodes and their responses. The second type of BitTorrent traffic is P2P file transfer traffic among BitTorrent nodes. Similar to the HTTP traffic, each P2P connection includes both headers (control message) and bodies (data message). Headers are used to exchange control information among peers. For example, a downloader node can tell its peers which portions of a file it needs. A message body always follows a header in a connection, with its length specified in the header.

To implement payload bypassing for BitTorrent traffic, *Nebula* needs to identify message bodies in BitTorrent P2P connections. Because these P2P connections do not use well-known port numbers, they can be identified only through their contents. For example, the first several bytes of a BitTorrent P2P connection includes a string "BitTorrent". By searching for this string in the beginning of every network connection, *Nebula* can reliably track BitTorrent P2P connections. After a BitTorrent P2P connection is identified, its message body part can be readily deduced based on the body length in the header. If other packets happen to have the "BitTorrent" string, their body length field may not match the packet length, which suggests the associated connection is not a BitTorrent connection.

3.5.4 eDonkey

There are also two types of eDonkey traffic, just like BitTorrent. The first type is between an eDonkey node and an eDonkey server, and does not carry files. An eDonkey server tracks which files are stored on which nodes. EDonkey nodes report to eDonkey servers the files they have and query eDonkey servers which eDonkey nodes have the files that they want. Unlike BitTorrent's tracker node, an eDonkey server can further search files based on keywords. Only P2P connections among eDonkey nodes carry file traffic. In each P2P connection, information is exchanged in a TLV-like (Type, Length, Value) structure, and the transferred files are encapsulated in a message type called "send part."

To implement payload bypassing for eDonkey traffic, *Nebula* first identifies eDonkey P2P connections, which always start with a sequence of "Hello" message, "file request" message, "slot request" message, and "request parts (of a file)" message. The overhead to scan these messages is small because each structure contains a length field to identify the next structure. Thus no string matching is required. Transferred files are always in the "send part" message. Therefore, after locating the "send part" message, the file data stream in the message can be simply skipped.

3.6 Attack Analysis

Although *Nebula*'s signature can successfully catch all existing buffer overflow attacks to Linux machines that we have found so far. Future attacks may attempt to evade *Nebula*'s detection logic and thus result in false negatives. There are several known weaknesses of *Nebula* and we discuss them in more detail in this section.

Nebula assumes the attack string contains at least N stack addresses or library function entry point addresses, where N is 3 by default. It is conceivable that future attacks may repeat the overwriting stack address only once or twice just to avoid *Nebula*'s detection. Fortunately, with the help of payload bypassing, *Nebula* may set N to 1 and still is able to detect most of these new attacks without generating many false positives. But payload bypassing has its own problem: it may create false negatives because some buffer overflow attacks indeed target at downloaded payload files.

Payload bypassing assumes there is no buffer overflow attack in the payload. But files being downloaded as payloads sometimes can indeed overflow some buffer of their viewer software. For example, some special crafted PNG file and JPG file can overflow some versions of Microsoft Internet Explorer. Fortunately, this type of attack requires user intervention to take effect and is thus considered a passive attack. In other words, the attacker cannot actively infect a user machine like worms. Therefore, although payload bypassing may miss some passive buffer overflow attacks, it will not miss any active buffer overflow attacks that are used by worms to propagate themselves from machine to machine as quickly as possible.

Finally, the number of false positives generated by *Nebula* when sanitizing traffic to Windows platforms are high, because Windows processes utilize a different memory layout than Linux ones. In a Windows-based buffer overflow attack [24], attackers change the return address to the address of an indirect jump instruction whose associated register has already been tampered with through other means. And the steppingstone indirect jump instruction can be anywhere in the DLLs or application code. In our future work, we plan to address this issue by developing new signatures for Windows platforms.

4. EVALUATION

To evaluate *Nebula*'s effectiveness in detecting buffer overflow attacks, we measure the false negative rate using input traffic containing known attacks, and the false positive rate under known innocent traffic. Then we measure the performance overhead of

Name	No. of Repetitions	Hijack Destination Address
LFTP Remote Stack-Based Overflow	24	0xbffff100
ATPHTTPd Buffer Overflow	> 30	0xbffff600
in.telnetd tgetent buffer overflow	> 30	0xbffff5d2
Samba Remote buffer overflow	> 30	0xbffff244
imapd remote overflow	> 30	0xbffff501
Remote INND buffer overflow exploit	19	0xbffff5d0
Remote vulnerability in LCDproc 0.4	4	0xbffff750
Tcpdump remote root vulnerability	10	0xbffff248
BSD Termcap overflow	> 30	0xbffff5c2
PoPToP PPTP Server Remote Exploit	> 30	0xbffff600

Table 1: *Analysis of ten publicly available Linux-based buffer overflow attacks, in terms of the number of times an attack pattern is repeated and the hijack destination address contained in the attack packet. All these attacks are CI attacks. Their measured stack sizes are all less than 16Kbytes.*

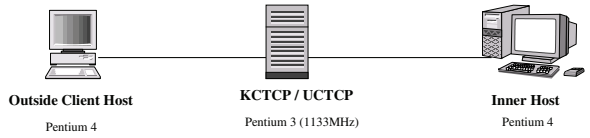

Figure 2: *The testbed set-up used to evaluate the effectiveness a performance of the CTCP architecture. All test hosts are equipped with an Intel Pro/1000 Gigabit Ethernet NIC.*

Nebula with and without payload bypassing. Figure 2 shows the test setup used in these tests. In this setup, both external client machines and internal server machines are Pentium-4 machines, whereas the CTCP router is a Pentium-3 (1133MHz) machine. All three machines are equipped with an Intel Pro/1000 gigabit NIC and run Linux 2.4.7.

4.1 False Negatives

For each of the 10 buffer overflow attacks listed in Table 1, we first set up a proper operating environment to collect the corresponding attack packet sequence. Then we applied the proposed CI and RTL attack signatures to each of the ten attack packet sequences, and the result is that *Nebula* can successfully detect all 10 attacks successfully. This result is not particularly surprising as none of existing buffer overflow attacks are designed to evade *Nebula*. However, the fact that *Nebula* can use a single signature to detect all of them demonstrates that the idea of attack-independent signature is indeed feasible.

4.2 False Positives

To test *Nebula*'s false positive rate, we used two types of packet payloads. *Static* samples are a collection of files stored on user hosts, including object files (such as library files and executable files), document files (such as pdf, ps, doc, txt, and HTML), picture files (such as gif, jpg, and mpeg), text files (such as HTML and txt), and gzip files, whose size is 348 Mbytes, 334 Mbytes, 220 Mbytes, 269 Mbytes, and 79 Mbytes, respectively. *Dynamic* samples are actual packet traces collected by sniffing the traffic on a 100-Mbps link that connects a university research lab to the Internet. Packets belonging to the same TCP connection are assembled together and stored in a separate file. Therefore each of these dynamic sample files records the whole conversation between the two communicating hosts of the corresponding TCP connection. We collected one month's worth of packet trace, which includes 134966 TCP connections and about 1.582 Gbytes of data. We ran these two samples through *Nebula*'s signature-matching algorithm, and measured the number of matches while varying the stack size and the minimal number of repetitions of the attack pattern. Every such match is considered as a false positive.

Repeat	2Mbyte	16Kbyte	8Kbyte
1	426260/1155877	1672/6157	1240/4195
2	202/176	1/22	1/11
3	37/92	1/13	1/5
10	2/16	0/4	0/0

Table 2: *Number of false positives under the static sample as reported by* Nebula. *The minimal number of times the attack pattern is repeated is assumed to be 1, 2, 3 or 10, and the stack size tested is 2Mbytes, 16Kbytes, or 8Kbytes. In each entry the left is the number of false positives for RTL attacks, whereas the right is the number of false positives for CI attacks.*

Repeat	2Mbyte	16Kbyte	8Kbyte
1	902265/2121784	5033/13928	2810/7648
2	214/77	0/14	0/0
3	43/21	0/10	0/0
10	1/2	0/1	0/0

Table 3: *Number of false positives under the dynamic sample as reported by* Nebula. *The minimal number of times the attack pattern is repeated is assumed to be 1, 2, 3 or 10, and the stack size tested is 2Mbytes, 16Kbytes, or 8Kbytes. In each entry the left is the number of false positives for RTL attacks, whereas the right is the number of false positives for CI attacks.*

Table 2 shows the result of the static sample test. When the minimal number of attack pattern repetition is 3 or above, the false positive rate is generally acceptable, regardless of the stack size. When the number of repetitions is larger than or equal to 2 and the stack size is smaller than or equal to 16Kbytes, all false positives are due to object files. Moreover, one object file could cause more than one false positives.

Table 3 shows the result of the dynamic sample test. When the stack size is 16Kbytes or smaller and the minimal number of attack pattern repetitions is 2 or 3, all false positives are caused by 4 out of a total of 134966 TCP connections traced, or a false positive rate of 0.00296%. These four TCP connections are used to transfer four different HTML page, among which three contain a gif file and one contains a jpg file.

Because the stack addresses in Linux must start with $0xbf$, which is not a printable ASCII character, and therefore rarely appears in telnet sessions, e-mail bodies, HTML text files, etc. Instead, it is more likely to show up in executable files, picture files, and files containing Unicode characters. Because the static sample contains more executable files, the false positive rate of the static sample test is higher than that of the dynamic sample test.

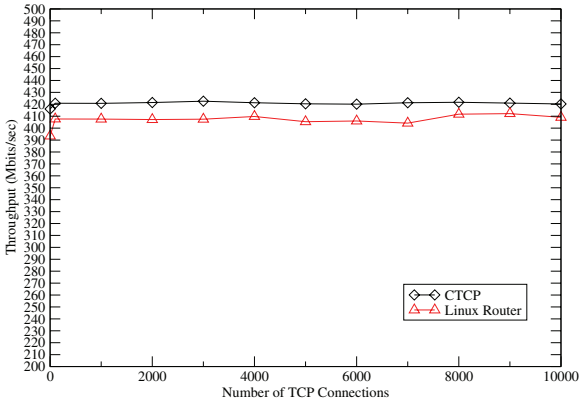

Figure 3: *Throughput comparison between a CTCP router and a Linux router, when the number of TCP connections increases from 1 to 10000.*

4.3 Performance Overhead

In performance overhead test, we measured the throughput of a CTCP router and a generic Linux router under different numbers of TCP connections between the client and the server, based on the testbed set-up shown in Figure 2. In each throughput measurement, the client host continues pumping data into every TCP connection. Results in Figure 3 show that the CTCP router actually provides a better throughput than the vanilla Linux router. In addition, CTCP's throughput is not affected by the number of TCP connections traversing through it, up to 10000 connections.

The throughput gain of CTCP over vanilla Linux mainly comes from two sources. First, under the CTCP architecture, the round-trip delay of each packet is shorter, because it is the CTCP router that creates the ACK packet, not the server, as in the generic Linux router case. As a result, the client is able to to send packets at a faster rate than it does under a Linux router, Second, the CTCP architecture allows the processing of a packet and the transmission of the next packet to proceed simultaneously. In other words, when the client checks the validity of an ACK packet of a previously sent packet, say P_1, and prepares for the next outgoing packet, say P_2, the CTCP router could process P_1 simultaneously.

To test the throughput penalty introduced by *Nebula* on a CTCP router, we measured its throughput with and without *Nebula* enabled, using the same testbed set-up shown in Figure 2. In these measurements, different numbers (ranging from 100 to 10000) of TCP connections between the client and the server are established before the client pumps data into each TCP connection. The throughput of each test is measured at the server host.

Experimental results show the CTCP router's throughput is 420 Mbits/sec when *Nebula* is disabled, and drops to 248 Mbits/sec when when *Nebula* is turned on. The performance degradation mainly comes from the additional processing overhead associated with examining every incoming byte.

4.4 Effectiveness of Payload Bypassing

We examined the effectiveness of payload bypassing from three angles: (1) What is the percentage of traffic that is "payload"? (2) How many false positives are eliminated? and (3) How much throughput improvement can be achieved? The same testbed as in Figure 2 is used. We tested four protocols, HTTP, FTP, BitTorrent, and eDonkey on Windows XP, to answer the first two questions. We use Windows applications because most Internet traffic is generated from Windows machines and programs using these protocols are readily available on the Windows platform. Files being downloaded are from the static sample, which has 8068 files with a total size of 1.22 Gbytes. We used Linux machines to answer the third question because we found that Windows XP can only achieve around 200 Mbps TCP throughput when two Windows XP machines are directly connected, whereas Linux can achieve 500 Mbps with the same set-up. Files used in this test are a random subset of the static sample because we can only afford to use a 500-Mbyte RAM disk for network throughput test. Using physical hard disks in this test is unacceptable because we were using gigabit Ethernet link.

Protocol	Control Traffic (Mbytes)	Control (%)	Payload (%)
HTTP	4.700	0.39%	99.61%
FTP	2.071	0.17%	99.83%
BitTorrent	3.984	0.33%	99.67%
eDonkey	4.347	0.36%	99.64%

Table 4: *Percentage of payload in the traffic when each of the four protocols that* Nebula *can recognize is used to transfer files of a total size of 1.22 Gbytes*

Protocol	False Positive Repeat = 1	False Positive Repeat = 3
HTTP	0	0
FTP	0	0
BitTorrent	0	0
eDonkey	49	0
No payload bypassing	426260+1155877	37+92

Table 5: *The number of false positives in the test traffic associated with different protocols after applying payload bypassing is negligible even when the attack pattern repetition count is 1.*

4.4.1 Percentage of Payload Traffic

The percentages of "payload" in the traffic when using each of the four protocols to transfer files in the static sample are shown in Table 4. To transfer around 1.22 Gbytes of data, the extra bytes introduced by each of the four protocols is only around 2 Mbytes to 5 Mbytes. This means that if payload bypassing is used, less than 0.4% of the traffic needs to be checked for buffer overflow attack, and more than 99.6% of the traffic can be safely ignored.

4.4.2 Effectiveness on Eliminating False Positive

The effectiveness of payload bypassing on eliminating false positive is shown in Table 5. The stack size used in the test is 2 Mbytes, the worst case. Before applying payload bypassing, when the attack pattern repetition count is 1, there are millions of false positives. With payload bypassing, text-based protocols such as HTTP and FTP have zero false positive regardless of the attack pattern repetition count and the content of transferred files. For binary-based protocols, only eDonkey protocol has a small number of false positives when the attack pattern repetition count is 1. This is not surprising as the control traffic of the eDonkey protocol accounts for only around 4 Mbytes.

4.4.3 Throughput Improvement

Only HTTP protocol is tested in this experiment because the computation overhead of HTTP payload bypassing is the highest among all four protocols. HTTP payload bypassing needs to search special string "Content-Length", "Transfer-Encoding", empty line, etc. in the HTTP header of each file transfer transaction. While FTP also needs to search special strings in the control connection, FTP control traffic is less than half of HTTP headers. For binary-based protocols, no search operation is required, as payloads reside at particular offsets of the data stream.

In the baseline case, the CTCP router supports neither buffer overflow attack nor payload bypassing, and its throughput is shown

Test Cases	Throughput (Mbps)
Direct connection	507
Linux router	409
CTCP without BO check	420
CTCP with BO check	248
CTCP with BO check and payload bypassing	420

Table 6: *The throughput of the CTCP router under a test HTTP connection when different options are turned on. With payload bypassing, a CTCP router can perform buffer overflow (BO) attack detection and still achieve a throughput higher than a generic Linux router.*

in Table 6. When two computers are connected directly, i.e., without any router in between, the HTTP throughput can reach around 500 Mbps assuming files are stored on RAM disks. Adding a Linux router or a CTCP router in between, the throughput is decreased to around 409 Mbps and 420 Mbps, respectively. When the buffer overflow check function in the CTCP router is turned on, its throughput is further decreased to 250 Mbps, because the CTCP router needs to examine every byte going through it. However, when the CTCP router enables payload bypassing, its throughput comes back up to 420 Mbps, as if buffer overflow attack detection costs nothing. This throughput gain arises because the CTCP router only needs to check the header parts of the tested HTTP traffic, which corresponds to a very small percentage of bytes that actually go through the CTCP router.

5. CONCLUSION

Buffer overflow attack is arguably the most dangerous attack method used today because new network applications continue to exhibit this type of vulnerabilities and many INTERNET worms use it to propagate themselves from machine to machine. Although many host-based solutions to buffer overflow attacks already exist, so far their impact on improving enterprise IT security is relatively limited. The main reason is that these solutions typically require disruptive changes to the existing IT infrastructure and therefore do not offer a feasible migration path that IT architects can reasonably take. This paper proposes a scalable network-based buffer overflow attack detection system called *Nebula*, which does not require any infrastructure modification, features a generalized buffer overflow attack signature that is able to detect all known variants of buffer overflow attacks, and exploits various contextual information to reduce the number of false negatives to a negligible level. Although existing network-based intrusion detection systems can also detect some buffer overflow attacks, the difference is that *Nebula* uses a single signature to detect all buffer overflow attack instances that are derived from the same principles of operation, zero-day or not. In addition, *Nebula* is built on a centralized CTCP architecture that can defeat all existing NIDS evasion techniques and incorporates a payload bypassing mechanism that can scale its signature matching logic up to gigabit/sec links on general-purpose server hardware. Experiments on a fully working *Nebula* prototype demonstrate its overall effectiveness in buffer overflow attack detection and its negligible run-time performance overhead.

Although the current *Nebula* prototype still cannot detect all possible buffer overflow attacks, it represents an important step toward effective network-based detection of buffer overflow attacks, including zero-day ones. We are currently incorporating a binary disassembler to the *Nebula* prototype so as to reduce the number of false positives when the attack pattern repeat count is set to 1. In addition, we are developing an implementation framework that can quickly import the protocol recognition logic added to Ethereal without manual programming. This implementation framework allows *Nebula* to leverage the development efforts behind Ethereal and increase the number of protocols its payload passing mechanism can recognize for free.

6. REFERENCES

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

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

[3] 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

[4] 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.

[5] Fu-Hau Hsu and Tzi-cker Chiueh, "CTCP: A Transparent Centralized TCP/IP Architecture for Network Security," Annual Computer Security Application Conference (ACSAC 2004), Tucson, Arizona, Dec., 2004.

[6] 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.

[7] 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.

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

[9] Ethereal: A Network Protocol Analyzer, www.ethereal.com

[10] FastTrack Description, http://www.p2pwatchdog.com/packet_fasttrack.html

[11] Manish Prasad, Tzi-cker Chiueh, "A Binary Rewriting Defense against Stack based Buffer Overflow Attacks," Usenix Annual Technical Conference, General Track, San Antonio, TX, June 2003

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

[13] 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).

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

[15] Stig Andersson, Andrew Clark, and George Mohay," Network-Based Buffer Overflow Detection by Exploit Code Analysis," AUSCERT 2004

[16] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "Operating system enhancements to prevent the misuse of system calls," Proceedings of the 7th ACM conference on Computer and Communications Security, 2000, Athens, Greece.

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

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

[19] Vendicator, “Stack Shield,” http://www.angelfire.com/sk/stackshield/

[20] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken, “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities,” NDSS 2000.

[21] CacheLogic, http://www.cachelogic.com/research/slide1.php

[22] C. Kruegel, T. Toth, and E. Kirda, “Service Specific Anomaly Detection for Network Intrusion Detection,” InSymposium on Applied Computing (SAC), Spain, March 2002.

[23] Ke Wang and S. J. Stolfo, “Anomalous Payload-based Network Intrusion Detection ,”Recent Advance in Intrusion Detection (RAID), Sept. 2005.

[24] DilDog, “The Tao of Windows Buffer Overflow,” http://www.cultdeadcow.com/cDc_files/cDc-351/index.html