Accurate Buffer Overflow Detection via Abstract Payload Execution

Thomas Toth and Christopher Kruegel

Distributed Systems Group
Technical University Vienna
Argentinierstrasse 8, A-1040 Vienna, Austria
{ttoth, chris}@infosys.tuwien.ac.at

Abstract. Static buffer overflow exploits belong to the most feared and frequently launched attacks on todays Internet. These exploits target vulnerabilities in daemon processes which provide important network services. Ever since the buffer overflow hacking technique has reached a broader audience due to the Morris Internet worm [20] in 1988 and the infamous paper by AlephOne in the `phrack` magazine [1], new weaknesses in many programs have been discovered and abused.
Current intrusion detection systems (IDS) address this problem in different ways. Misuse based systems attempt to detect the signature of known exploits in the payload of the network packets. This can be easily evaded by a skilled intruder as the attack code can be changed, reordered or even partially encrypted. Anomaly based network sensors neglect the packet payload and only analyze bursts of traffic thus missing buffer overflows altogether. Host based anomaly detectors that monitor process behavior can notice a successful exploit but only a-posteriori when it has already been successful. In addition, both anomaly variants suffer from high false positive rates.
In this paper we present an approach that accurately detects buffer overflow code in the packet's payload by concentrating on the *sledge* of the attack. The sledge is used to increase the changes of a successful intrusion by providing a long code segment that simply moves the program counter towards the immediately following exploit code. Although the intruder has some freedom in shaping the sledge it has to be executable by the processor. We perform abstract execution of the payload to identify such sequences of executable code with virtually no false positives.
A prototype implementation of our sensor has been integrated into the `Apache` web server. We have evaluated the detection rates as well as the performance impact of our proposed system.

Key words: Intrusion Detecion, Buffer Overflow Exploit, Network Security

1 Introduction

The constant increase of attacks against networks and their resources causes a necessity to protect these valuable assets. Although well-configured firewalls provide good protection against many attacks, some services (like `HTTP` or `DNS`) have to be publicly available. In such cases a firewall has to allow incoming traffic from the Internet without restrictions. The programs implementing these services are often complex and old pieces of software. This inevitably leads to the existence of programming bugs. Skilled intruders exploit such vulnerabilities by sending packets with carefully crafted content that overflow a static buffer in the victim process. This allows the intruder the altering of the execution flow of the service daemon and the execution of arbitrary code that he can inject, eventually leading to a system compromise and elevating the privileges of the attacker to the ones of the service process. Such an attack is called a *buffer overflow exploit*. Studies [18] have indicated that these attacks contribute to a large number of system compromises as most daemons run with `root` privileges.

Intrusion detection systems (IDS) are security tools that are used to detect traces of malicious activities which are targeted against the network and its resources. IDS are traditionally classified as anomaly or signature based. Signature based systems like Snort [17] or NetStat [22, 23] act similar to virus scanners and look for known, suspicious patterns in their input data. Anomaly based systems watch for deviations of actual from expected behavior and classify all 'abnormal' activities as malicious.

As signature based designs compare their input to known, hostile scenarios they have the advantage of raising virtually no *false alarms* (i.e. classifying an action as malicious when in fact it is not). For the same reason, they have the significant drawback of failing to detect variations of known attacks or entirely new intrusions.

Because of the ability to detect previously unknown intrusions a number of different anomaly based systems have been proposed. Depending on their source of input data, they are divided into host based and network based designs.

Host based anomaly detection systems can focus on user or program behavior. User profiles are built from login times and accessed resources [5, 10, 2] (e.g. files, programs) or from timing analysis of keystrokes [19]. Unfortunately, user behavior is hard to predict and can change frequently. Additionally, such systems cannot react properly when network services get compromised as no single user profile can be associated to a daemon program.

As a consequence the focus was shifted from user to program behavior. The execution of a program is modeled as a set of system call sequences [7, 6] which occur during 'normal' program execution. When the observed sequences deviate from the expected behavior the program is assumed to perform something unintended, possibly because of a successful attack (e.g. buffer overflow). Other researchers use neural networks [9] and concentrate on the analysis of the input data that is passed to programs. These systems are capable of detecting buffer overflows attacks against service daemons but only *after* they have been successful and manifest themselves in abnormal behavior. This has the problem that damage might already have occurred. We present a system that analyzes the content of service requests at the network level and can a-priori prevent malicious code from being executed.

This is similar to network based anomaly detection systems which do not concentrate on activities at hosts (e.g. users or programs) but focus on the packets that are sent over the network. Current network based systems [15, 16, 4, 21] however only model the flow of packets. The source and destination IP addresses and ports are used to determine parameters like the number of total connection arrivals in a certain period of time, the inter-arrival time between packets or the number of packets to/from a certain machine. These parameters can be used to reliable detect port scans or denial-of-service (DOS) attempts. Unfortunately, the situation changes when one considers buffer overflow attacks. In this case the attacker sends one (or a few) packets including the attack code which is executed at the remote machine on behalf of the intruder to elevate his privileges. As the attacker only has to send very few packets (most of the time a single one is sufficient), it is nearly impossible for systems that use traffic models to detect such anomalies.

We propose an approach to do buffer overflow detection at application level for important Internet services. These services usually operate in a client/server setup where a client machine sends a request to the server which returns a reply with the results. Our detection approach distinguishes normal request data from malicious content (i.e. buffer overflow code) by performing abstract execution of the payload data contained in client requests. In the case of detecting long 'instruction chains' of executable code (see Section 3) a request can be dropped before the malicious effects of the exploit code are triggered within vulnerable functions (and maybe detected by another ID system afterwards).

The first section describes buffer overflow exploits in general and the possibilities of an attacker to disguise his malicious payload to evade ID systems. Then we present our approach of abstract payload execution and explain the advantages of this design. The following section introduces the results of the integration of our prototype into the `Apache` [3] web server. Finally, we briefly conclude and outline future work.

2 Buffer Overflow Exploits

Many important Internet services (e.g. `HTTP`, `FTP` or `DNS`) have to be publicly available. They operate in a client/server setup which means that clients send request data to the server, which operates on the given input and returns a reply containing the desired results or an error message. This means that virtually everyone (including people with malicious intents) can send data to a remote server daemon which has to analyze and process the presented data.

The server daemon process usually allocates memory buffers where request data received from the network by the underlying operating system is copied into. During the handling of the received data, the input is parsed, transformed and often copied several times. Problems arise when data is copied into fixed sized buffers declared in subroutines that are statically allocated on the process' stack. It is possible that the request that has been sent to the service is longer than the allocated buffer. When the length of the input is not checked and copied into the buffer by means of an *unsafe* C string function parts of the stack that are adjacent to the static buffer may be overwritten - a *stack overflow* occurs.

Unsafe C library string functions (see Table 1 for examples) are routines that are used to copy data between memory areas (buffers). Unfortunately, it is not guaranteed that the amount of data specified as the source of the copy instruction will fit into the destination buffer. While some functions (like strncpy) at least force the programmer to specify the number of bytes that should be moved to the destination, others (like strcpy) copy data until they encounter a terminating character in the source buffer itself. Nevertheless, neither functions check the length of the destination area.

strcpy	wstrcpy	strncpy	wstrncpy
strcat	wcscat	strncat	wstrncat
gets	getws	fgets	fgetws
sprintf	swprintf	scanf	wscanf
memcpy	memmove		

Table 1. Vulnerable C Library Functions

Especially functions that determine the end of the source buffer by relying on data inside that buffer carry a risk of overflowing the destination memory area. This risk is especially high when the source buffer contains unchecked data directly received from clients as it allows attackers to force a stack overflow by providing excessive input data.

The fact that C compilers (like gcc [8]) allocate memory for local variables (including static arrays) as well as information which is essential for the program's flow (the return address of a subroutine call) on the stack makes static buffer overflows dangerous. Figure 1 below shows the stack layout of a function compiled by gcc. When an attacker can overflow a local buffer stored on the stack and thereby modify the return address of a subroutine call, this might lead to the execution of arbitrary code on behalf of the intruder.

An adversary that knows that a subroutine in the daemon process utilizes a vulnerable function (e.g. strcpy) can launch an attack by sending a request with a length that exceeds the size of the statically allocated buffer used as the destination by this copy instruction. When the server processes his input, a part of the stack including the subroutine's return address is overwritten (see Figure 2 below). When the attacker simply sends garbage, a segmentation violation is very likely to occur as the program continues at a random memory address after returning from the subroutine.

A skillful attacker however could carefully craft his request such that the return address points back to the request's payload itself which has been copied onto the stack into the destination buffer. In this case the program counter is set to the stack address somewhere in the buffer that has been overflown when the

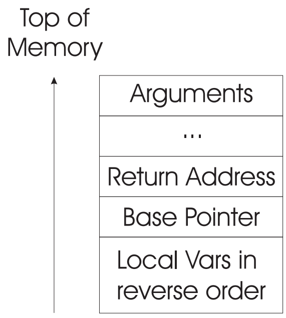

Fig. 1. Stack Layout

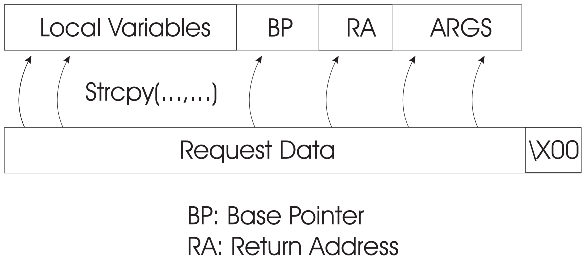

Fig. 2. Operation of `strcpy(dst *, src*)`

subroutine returns. The processor then resumes execution of the bytes contained in the request with the privileges of the server process (often with `root` rights).

The main problem with this technique is the fact that the attacker does not know the exact stack address where his payload will be copied to. Although the intruder can compile and analyze the service program on his machine to get a rough idea of the correct address, the exact value depends on the environment variables that the user has set. When a wrong address is selected, the processor will start to execute instructions at that position. This is likely to result in an illegal opcode exception because the random value at this memory position does not represent a valid processor instruction (thereby killing the process). Even when the processor can decode the instruction, its parameter may reference memory addresses which are outside of any previously allocated areas. This causes a segmentation violation and a termination of the process.

To circumvent this problem, the attacker can put some code in front of the exploit code itself to increase the chances of having the faked return address point into the correct stack region. This extra code is called the *sledge* of the exploit and is usually formed by many (a few hundreds are common) `NOP` (no operation) instructions. The idea is that the return address simply has to point somewhere into this long sledge that does nothing except having the processor move the program counter towards the actual exploit code. Now it is not necessary anymore to hit the exact beginning of the exploit code but merely a position somewhere in the sledge segment. After the exploit code, the guessed return address (RA) (which points into the sledge) is replicated several times to make sure that the subroutine's real return address on the stack is overwritten. A typical layout of a buffer overflow code that includes a sledge is shown in Figure 3.

Fig. 3. Typical Structure of a Buffer Overflow Exploit

Some network based misuse IDS (like Snort [17]) try to identify buffer overflow exploits by monitoring the network traffic and scanning packet payload for the occurrence of suspicious bytecode sequences. These sequences are drawn from actual exploits and represent strings like `/bin/sh` or operating system calls.

This suffers from the problem that there are virtually infinite variations of buffer overflow exploits that attack different vulnerabilities of the same service or express the same functionality differently. In addition, code transformation techniques like reordering or insertion of filling instructions change the signature of the exploit and render the misuse based detection useless. Some intruders have even started to encode the actual exploit with a simple routine (e.g. `ROT-13`) while placing the corresponding decode routine in front of the encrypted exploit. When the buffer overflow is executed the decode routine first decrypts the exploit segment and then executes it.

The wide variety of different exploit signatures shifted the focus of these systems to the sledge. Every attack includes a long chain of architecture specific NOP (no operation) instructions that precedes the actual exploit - `0x90` for the `Intel IA32` [11], for other architectures refer to [14].

Unfortunately, the sledge of a buffer overflow exploit can also use different opcodes than a NOP causing the signature detectors to fail when these instructions are replaced by functionally equivalent ones. According to [12] there are more than 50 opcodes for the `Intel IA32` architecture which are suitable for replacing the NOP operation (Table 2 below enumerates a few examples).

Using these operations (or any combination of them) causes the sledge to behave exactly the same as before, nevertheless its shape can be modified to evade misuse based ID systems. Notice that it is not possible to encrypt the sledge because it has to be executed before the exploit code (and any decryption routines).

The operations presented in Table 2 behave exactly like the NOP instructions on an `Intel` architecture in the sense that they are only a single byte long. By considering the fact that modern compilers align variables and data structures on the stack at word boundaries[1], more sledge modifications can be performed.

[1] Word alignment means that the address of any variable allocated on the stack modulo four equals 0

Mnemonic	Explanation	Opcode
AAA	ASCII Adjust After Addition	0x37
AAS	ASCII Adjust After Subtraction	0x3f
CWDE	Convert Word To Doubleword	0x98
CLC	Clear Carry Flag	0xf8
CLD	Clear Direction Flag	0xfc
CLI	Clear Interrupt Flag	0xfa
CMC	Complement Carry Flag	0xf5
...	...	...

Table 2. Single-Byte NOP Substitutes for `IA32`

Instead of using single byte instructions without parameters, an intruder can even use multi-byte opcodes with arguments. One just has to make sure that executable code is present at all positions starting at word boundaries. This is necessary because the return address could point to the beginning of any word (i.e. 4 bytes) inside the sledge. This allows the attacker to choose assembler instructions like the ones depicted in Table 3 below or any similar to them. Operations that require an immediate parameter are best suited for this because there is no risk of accessing illegal memory areas (thereby creating a segmentation violation). The attacker just has to use a return address that also points to a word aligned boundary.

Instruction	Bytecode
adc $0x70,%cl	80 d1 70
adc $0x70d18070, %ecx	81 d1 70 80 d1 70
and $0x55120125, %eax	25 01 12 44
jmprel 0x37	0xeb 0x37
...	...

Table 3. Multi-Byte NOP Substitutes for `IA32`

The only restriction that remains for the intruder when creating the exploit code and the sledge is that no `NULL` (0x00) characters may be present. This is due to the fact that a `NULL` character is interpreted as the end character by many vulnerable `C` functions. Because the complete attack code has to be copied, the routines may not terminate prematurely. Other than that, the intruder has virtually no limitations in designing his exploit.

Any network based misuse system can be easily evaded when such a freedom is given in choosing the layout of the attack. Even anomaly based systems [13] that base their analysis on the payload of the packet could be fooled. Such systems operate with profiles of a 'normal' request that can be imitated with the means shown above. Network based anomaly detectors that consider only protocol information or the flow of traffic fail as well because only a few legal packets need to be transmitted.

While host based anomaly sensors can notice the effect of a completed buffer overflow and raise an appropriate alarm, this approach is undesirable because of two reasons. First, only successful attacks which cause a corresponding distortion in process behavior are reported. No indication on the number of attempts is given. Second, the system can only react to the attack after it has manifested itself as weird behavior. Potential damage that has been inflicted before the ID can react cannot be prevented (e.g. deletion/modification of files).

We present a system that accurately detects exploits in the payload of application requests and is capable of stopping malicious activity before it effects the system even when the attacker applies all evasion mechanisms described above. The idea of our approach is to focus on the executability of the sledge (which cannot be prevented without breaking the attack) by means of abstract execution.

3 Abstract Execution

The following two properties that classify a sequence of bytes executed on behalf of a certain process are used to define *abstract execution*.

- **Correctness:** A sequence of bytes is correct, if it represents a single valid processor instruction. This implies that the processor is able to decode it. The byte sequence consists of a valid opcode and the exact number of arguments needed for this instruction (but no more). Otherwise, the sequence is incorrect.
- **Validity:** A sequence of bytes is valid if it is correct and all memory operands of the instruction reference memory addresses that the process which executes the operation is allowed to access. A memory operand of an instruction is an operand that directly references memory (i.e. specifies an address in the process' memory area). Validity is important because references to non-accessible memory addresses will be detected by the operating system resulting in the immediate termination of the process with a segmentation violation. A correct instruction without any (memory) operands (e.g. NOP) is automatically valid. We also call a valid byte sequence a valid instruction.

Definition:

> A sequence of bytes is *abstract executable* if it can be represented as a sequence of consecutive valid instructions.

We partition the pool of processor instructions into two sets. One contains all instructions that alter the execution flow of a process (i.e. operations that modify the program counter) while the other one consists of the rest. The elements of the first set (e.g. `call`, `jmp`, `jne`) are called *jump* instructions. A sequence of valid instructions can be decomposed into subsequences that do not contain jump instructions. Such subsequences are called *valid instruction chains (IC)*. An instruction chain ends with a jump instruction or not abstract executable bytes. The length of an instruction chain is equal to the number of operations that it consists of.

An important metrics is the **execution length** of a sequence of valid instruction. The basic idea is that the execution length combines the lengths of instruction chains that are connected by jump instructions. It can be computed for a byte sequence using the following algorithm.

Algorithm: Execution Length of Byte Sequence

> The algorithm expects two input arguments, a byte sequence `seq` and a position `pos` in this sequence. It uses an auxiliary array `visited` to mark already visited blocks whose elements are initialized with false. The return value is a positive integer denoting the execution length starting at position `pos`.

1. When the instruction at `pos` is invalid, return 0.
2. When the instruction at `pos` has already been visited, a loop is detected and 0 returned.
3. Find the instruction chain starting at `pos` and calculate its length L. In addition, mark all its operations as visited.
4. When the instruction chain ends with invalid bytes, return L.
5. Otherwise, the chain ends in a jump instruction. When the target of the jump is outside the byte sequence `seq` or cannot be determined statically, return $L + 1$.
6. When the jump targets an operation at position `target` that is inside the sequence, call the algorithm recursively with the position set to `target` and assign the result to L'.
7. When the jump is an unconditional one, return $L + L'$.
8. Otherwise, it is a conditional jump. Call the algorithm recursively for the continuation of the jump - i.e. set the position to the operation immediately following the jump instruction and assign the result to L''. Then determine the maximum of L' and L'' and assign it to L_{max}. Then return $L + L_{max}$.

Definition:

> The *maximum executable length (MEL)* of a byte sequence is the maximum of all execution lengths that are calculated by starting from every possible byte position in the sequence. It is possible that a byte sequences contains several disjoint abstract execution flows and the MEL denotes the length of the longest.

4 Detecting Buffer Overflows

Following the definitions above, we expect that requests which contain buffer overflow exploits have a very high MEL. The sledge is a long chain of valid instructions that eventually leads to the execution of the exploit code. Even when the attacker inserts jumps and attempts to disguise the functionality of this segment, its execution length is high. In contrast to that, the MEL of a normal request should be comparatively low. This is due to the fact that the data exchanged between client and server is determined by the communication protocol and has a certain semantics. Although parts of that data may represent executable code, the chances that random byte sequences yield a long executable chain is very small.

The idea is that a static threshold can be established that separates malicious from normal requests by considering requests with a large MEL as malicious while those with a small execution length as regular. Because the sledge has to be executable in order to fulfill its task, a simple test is utilized to find long executable chains. Requests are analyzed immediately after they have been received by the server. This enables the system to drop potential dangerous packets before the service process can be affected and executes vulnerable functions.

The following observation allows an improvement of the search algorithm that has to determine the MEL of requests. According to the definition of the maximum executable length, all positions in the request's byte sequence could potentially serve as a starting point for the longest execution flow. However, when the MELs of normal requests and exploits differ dramatically, it is not necessary to search for the real maximum length. It is sufficient to choose only some random sample positions within the byte sequence and calculate the abstract executable length from these positions. Instructions that have been visited by earlier runs of the algorithm are obviously ignored. The rationale behind this improvement is the fact that it is very likely that at least one sample position is somewhere in the middle of the sledge leading to a tremendously higher MEL than encountered when checking normal requests.

5 Implementation

The algorithms to determine a single execution length and to choose reasonable sample points in the byte sequence of a request have been implemented in `C`. Because the recursive procedures are potentially costly, the main focus has been on an efficient realization. As every request needs to be evaluated, the additional pressure on the server must be minimized.

An important point is the decoding of byte sequences to determine the correctness and validity of instructions. We have chosen a static trie that stores all supported processor instructions together with the required operands and their types.

A trie is a hierarchical, tree like data structure that operates like a dictionary. The elements stored in the trie are the individual characters of 'words' (which are opcodes in our case). Each character (byte) of a word (opcode) is stored in the trie at a different level. The first character of a word is stored at the root node (first level) of the trie together with a pointer to a second-level trie node that stores the continuation of all the words starting with this first character. This mechanism is recursively applied to all trie levels. The number of characters of a word is equal to the levels needed to store it in a trie. A pointer to a leave node that might hold additional information marks the end of a word.

We store all supported opcodes of the processor's instruction set in the trie to enable rapid decoding of byte sequences. The leaf nodes hold information about the number of operands for each instruction together with their types (immediate value, memory location or register). This enables us to calculate the total length of the instruction at runtime by determining the necessary bytes for all operands.

It is important to notice that different instruction can be of different length, therefore a hash table is not ideally suitable. Currently, only the `Pentium` instruction set [11] has been stored in this trie, but no `MMX` and `SIMD` instructions are supported.

Figure 4 shows a simplified view of our trie. The opcodes for the instructions `AAA` (code 0x37) and `ADC` (code 0x80d1 - add with carry a one byte value to the `CL` register) have been inserted.

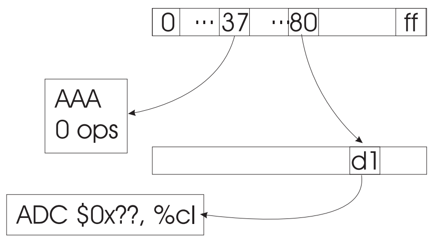

Fig. 4. Storing instruction opcodes in the trie

These algorithms are used to determine an approximation for the MEL of `HTTP` requests and have been integrated as a module into an `Apache 1.3.23` web server. During the startup of the server, the trie is filled and a function to check the request is registered as a `post_read_request` procedure. The `Apache` configuration file has been adapted to make sure that our module is the first to be invoked.

Each time a request arrives at the `HTTP` server, our subroutine calls the URL decoding routine provided by `Apache` and then searches for executable instructions in the resulting byte sequence. It is necessary to decode the request first to make sure that all escaped characters are transformed into their corresponding byte values. The module uses a definable threshold and stops the test immediately when a detected executable length exceeds this limit. The sampling rate is currently set to 30 points per kilobyte of request data.

6 Evaluation

6.1 Execution Length of HTTP Requests

In order to estimate the maximum executable length of regular HTTP requests, we calculated the MEL for service requests targeted at our institute's web server. Only successful requests that completed without errors have been chosen to make sure that the data set contains no packets with buffer overflow exploits. An additional ID system has been deployed to verify that assumption. 117228 server requests which we have been captured during a period of 7 days have been processed. The resulting MELs are shown in Figure 5 below.

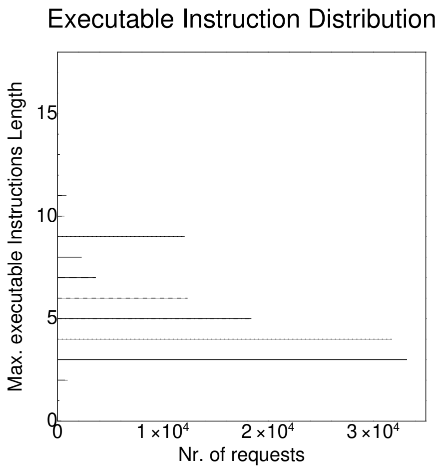

Fig. 5. Maximum Execution Length for regular HTTP Requests

Only 350 requests had a MEL value of 0 meaning that they did not contain a valid instruction at all. Most of the packets showed a maximum executable length of 3 and 4 (33211 and 31791 respectively) with the numbers decreasing for increasing lengths. The highest maximum instruction length that has been encountered was 16 which appeared for a total of 14 HTTP queries. As expected the numbers indicate that the MELs for regular requests are short.

6.2 Execution Length of Exploits

In order to support our claim that buffer overflow attacks contain long valid instruction chains, a number of available exploits have been analyzed. We have chosen buffer overflow attacks against the `Internet Information Service (IIS)` (the web server from `Microsoft`) and a `WU-FTP` exploit, all from [18]. Although our prototype has been tested with a web server, attack code against a different

Exploit	Max. Executable Length (MEL)
`IIS 4 hack 307`	591
`IIS 2000`	635
`wu-ftpd/2.6-id1387`	251

Table 4. Chain Length of Exploits

service daemon has been evaluated to show the applicability of our approach to other areas as well. The results of this evaluation are listed in Table 4.

According to the table above, the maximum executable lengths of requests that contain buffer overflow exploits is significantly higher than those of normal queries. This observation supports our assumptions presented in Section 4. For the actual detection of intrusions, a threshold has to be chosen to separate malicious from normal traffic and to raise an alert when the limit is exceeded.

For our first prototype, we simply select a 'reasonable' magic number between the maximum value gathered from the set of normal requests (16) and the minimum among the evaluated exploits (251). Because an attacker might attempt to limit the MEL by choosing a shorter sledge, the value should stay closer to the maximum of the normal requests. We decided to select 30 for the deployment of the probe. This leaves enough room for regular requests to keep the false positive rate low and forces an intruders to reduce the executable parts of his exploit to a length less than this limit to remain undetected. Such a short sledge nevertheless seriously impacts the attacker's chances to guess an address that is 'close enough' to the correct one to succeed.

6.3 Performance Results

To evaluate the performance impact of our module on the web server, we used the `WebSTONE` [24] benchmark provided by Mindcraft. `WebSTONE` can simulate an arbitrary number of clients that request pages of different sizes from the web server to simulate realistic load. It determines a number of interesting properties that are listed below.

- Average and maximum connection time of requests
- Average and maximum response time of requests
- Data throughput rate

The *connection time* is the time interval between the point when the client opens a `TCP` connection and the point when the server accepts it. The *response time* measures the time between the point when the client has established the connection and requests data and the point when the first result is received. The *data throughput rate* is a value for the amount of data that the web server is able to deliver to all clients.

The connection and response time values are relevant for the time a user has to wait after sending a request until results are delivered back. These times also characterize the number of requests a web server is able to handle under a specific load. The data throughput rate defines how fast data can be sent from the web server to the client. Because clients obviously cannot receive replies faster than the server is sending them, this number is an indication for how long a client has to wait until a request completes.

Our experimental setup consists of one machine simulating the clients that perform `HTTP` requests (`Athlon`, 1 Ghz, 256 MB RAM, `Linux 2.4`) and one host with the `Apache` server (`Pentium III`, 550 MHz, 512 MB RAM, `Linux 2.4`). Both machines are connected using a 100 Mb `Fast Ethernet`. `WebSTONE` has been configured to launch 10 to 100 clients in steps of 10, each running for 2 minutes. We did only a measurement of static pages, so no tests involved dynamic creation of results.

We measured the connection rate, the average client response time and the average client throughput for each test run with and without our installed module. The results are shown in Figures 6, 7 and 8. The dotted line represent the statistics gathered when running the unmodified `Apache` while the solid line represent the one with our activated module.

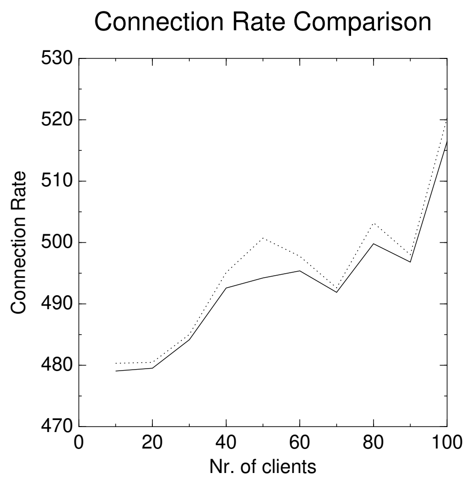

Fig. 6. Cient Connection Rate

As can be seen above, the connection rate has dropped slightly when our sensor is activated. The biggest difference emerged when 50 clients are active and a value of 494,2 connections per second versus 500,7 connections per second with the unmodified `Apache` has been observed. While this maximum difference is 6.5 connections per second (yielding a decrease in the client connection rate of about 1.4 %), the average value is only 2,4 (about 0.5 %).

There has been no significant decrease in the average response time. Both lines are congruent with regards to the precision of measurements.

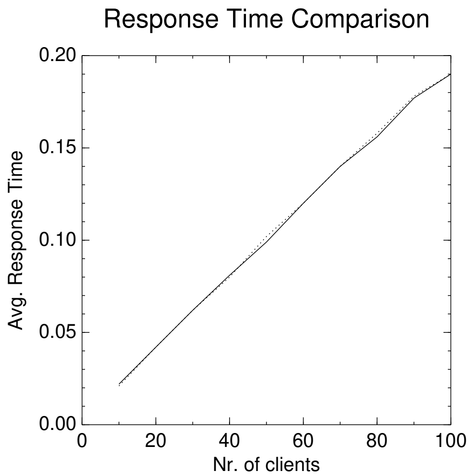

Fig. 7. Average Response Time

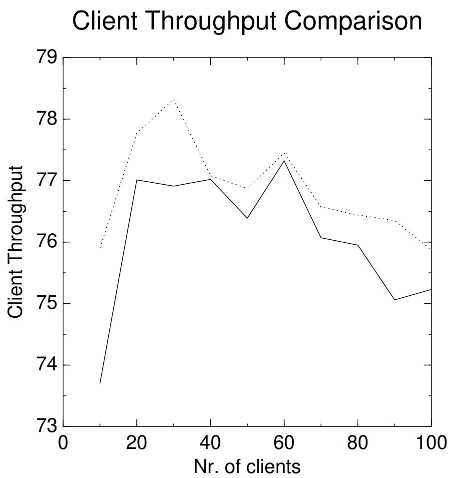

Fig. 8. Client Throughput

The client throughput decreased most with 10 active clients when it dropped from 75,90 Mbit per second to 73,70 MBit per second. This is an absolute difference of 2,2 MBit per second. ($\tilde{2}$,9 %). On average, the client throughput only decreased by 0,8 Mbit per second ($\tilde{1}$,05%).

The trie consumed about 16 MB of memory during the tests. While this seems to be a large number at first glance, one has to take the usual main memory equipment of web servers into account where a Gigabyte RAM is not uncommon. In addition, this data structure makes very fast tests possible and is a classical trade-off in favor of speed.

7 Conclusion and Future Work

This paper presents an accurate way to detect buffer overflow exploit code in Internet service requests. We explain the structure and constraints of these attacks and discussed methods used by intruders to evade common detection techniques.

Our analysis approach bases on the abstract execution of the packet payload to detect the sledge of an exploit. We define a valid instruction chain as a number of consecutive bytes in a request that represent executable processor instructions. The detection mechanism uses the fact that requests which contain buffer overflow code include noticeably longer chains than regular requests. In addition to the provision of theoretical support, our hypothesis has been verified by comparing the results for regular `HTTP` requests to ones with exploit code.

The system has the advantage that requests can be analyzed and denied a-priori before the service process is affected by a buffer overflow. It is also resistant to the presented evasion techniques in Section 2. The performance impact of the probe has been evaluated by integrating it into the `Apache` web server.

Further work will concentrate on emulating instructions that have not been included yet (`SIMD` and `MMX` operations). Additionally, we plan to collect experimental data for other protocols like `FTP` and `DNS` to validate that our proposed approach is also applicable there.

References

1. AlephOne. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1996.
2. Debra Anderson, Thane Frivold, Ann Tamaru, and Alfonso Valdes. *Next Generation Intrusion Detection Expert System (NIDES)*. SRI International, 1994.
3. The Apache Software Foundation. http://www.apache.org.
4. M. Bykova, S. Ostermann, and B. Tjaden. Detecting network intrusions via a statistical analysis of network packet characteristics. In *Proceedings of the 33rd Southeastern Symposium on System Theory*, 2001.
5. Dorothy Denning. An intrusion-detection model. In *IEEE Symposium on Security and Privacy*, pages 118–131, Oakland, USA, 1986.
6. Laurent Eschenauer. Imsafe. http://imsafe.sourceforge.net, 2001.
7. Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

8. The GNU Compiler Collection. http://gcc.gnu.org.
9. A. Ghosh and A. Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *USENIX Security Symposium*, 1999.
10. Judith Hochberg, Kathleen Jackson, Cathy Stallins, J. F. McClary, David DuBois, and Josephine Ford. Nadir: An automated system for detecting network intrusion and misuse. *Computer and Security*, 12(3):235–248, May 1993.
11. Intel. *IA-32 Intel Architecture Software Developers Manual Volume 1-3*, 2002. http://developer.intel.com/design/Pentium4/manuals/.
12. Home of K2. http://www.ktwo.ca.
13. Christopher Kruegel, Thomas Toth, and Clemens Kerer. Service Specific Anomaly Detection for Network Intrusion Detection. In *Symposium on Applied Computing (SAC)*. ACM Scientific Press, March 2002.
14. Mudge. Compromised: Buffer-Overflows, from Intel to SPARC Version 8. http://www.l0pht.com, 1996.
15. Peter G. Neumann and Phillip A. Porras. Experience with emerald to date. In *1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 73–80, Santa Clara, California, USA, April 1999.
16. Phillip A. Porras and Peter G. Neumann. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th NIS Security Conference*, October 1997.
17. Martin Roesch. Snort - lightweight intrusion detection for networks. In *USENIX Lisa 99*, 1999.
18. SecurityFocus Corporate Site. http://www.securityfocus.com.
19. Jude Shavlik, Mark Shavlik, and Michael Fahland. Evaluating software sensors for actively profiling windows 2000 computer users. In *Recent Advances in Intrusion Detection (RAID)*, 2001.
20. E. Spafford. The Internet Work Program: Analysis. *Computer Communication Review*, January 1989.
21. Stuart Staniford, James A. Hoagland, and Joseph M. , McAlerney. Practical automated detection of stealthy portscans. In *Proceedings of the IDS Workshop of the 7th Computer and Communications Security Conference*, Athens, 2000.
22. Giovanni Vigna and Richard A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. In *14th Annual Computer Security Applications Conference*, December 1998.
23. Giovanni Vigna and Richard A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.
24. WebSTONE - Mindcraft Corporate Site. http://www.mindcraft.com.