International Journal of Emerging Technology and Advanced Engineering
Website: www.ijetae.com (ISSN 2250-2459, ISO 9001:2008 Certified Journal, Volume 3, Issue 2, February, 2013)

A Signature-Free Buffer Overflow Attack Blocker Using Genetic Programming

Kotha Jothsna[1], Dr. R.V. Krishniah[2],
DRK Institute of Science and Technology, JNTUH, Hyderabad

***Abstract*— Now days internet threat takes a blended attack form, targeting individual users to gain control over networks and data. Buffer Overflow which is one of the most occurring security vulnerabilities in Internet services such as such as web service, cloud service etc. Motivated by the observation that buffer overflow attacks typically contain executables whereas legitimate client requests never contain executables in most Internet services. Unlike the previous detection algorithms, a new SigFree uses a Genetic Programming technique that is generic, fast, and hard for exploit code to evade. SigFree blocks attacks by detecting the presence of code, it is a signature free, thus it can block new and unknown buffer overflow attacks; SigFree is also immunized from most attack-side code obfuscation. To do so, we pay particular attention to the formulation of an appropriate fitness function and partnering instruction set. Moreover, by making use of the intron behaviour inherent in the genetic programming paradigm, we are able to explicitly Obfuscate the true intent of the code. All the resulting attacks Defeat the widely used in Intrusion Detection System.**

***Keywords*— Linear Genetic Programming, code injection, Intrusion Detection Systems.**

I. Introduction

The history of internet security, buffer over-flow is one of the most serious vulnerabilities in computer systems. Buffer overflow vulnerability is a root cause for most of the cyber attacks such as server breaking-in, worms, zombies, and bonnets. A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. This causes the data to overwrite into adjacent memory locations, and depending on what is stored there, the behaviour of the program itself might be affected. Although taking a broader viewpoint, buffer overflow attacks do not always carry binary code in the attacking requests (or packets); code-injection buffer overflow attacks such as stack smashing probably count for most of the buffers overflow attacks that have happen in the real world.

Although lot of research[2-10] has been done to tackle buffer overflow attacks, existing defences are still quite limited in meeting four highly desired requirements: 1) Simplicity in Maintenance; 2) Transparency to existing (legacy) server OS, application software, and hardware; 3) Resiliency to obfuscation; 4) Economical Internet-wide deployment. As a result, although several secure solutions have been proposed, they are not pervasively deployed, and a considerable number of buffer overflow attacks continue to be successful on a daily basis.

To overcome the above limitations, Recently, X. Wang et al.[11] proposed a SigFree, an online buffer overflow attack blocker, to protect Internet services. The idea of SigFree is motivated by an important observation that “the nature of communication to and from network services is predominantly or exclusively data and not executable code”. Their experimental study shows that the dependency-degree-based SigFree could block all types of code-injection attack packets tested in our experiments with very few false positives. Moreover, SigFree causes very small extra latency to normal client requests when some requests contain exploit code. However, Sig Free cannot fully handle self-modifying code and cannot fully handle the branch-function-based obfuscation. Further, Sig Free does not detect attacks such as return-to-libc attacks that just corrupt control flow or data without injecting code.

In this paper, we propose a novel SigFree, an online buffer overflow attack blocker using *Genetic Programming* (GP) [12] instead of *code abstraction* [11] to protect internet services. The idea of SigFree is motivated by an important observation that “the nature of communication to and from network services is predominantly or exclusively data and not executable code”. For blocking code-injection buffer overflow attack messages targeting at various internet services such as web service, Motivated by the observation that buffer overflow attacks typically contain executables whereas legitimate user requests never contain executables in most internet services.

The SigFree Fig. 1[11] works as follows: SigFree is an application layer blocker that typically stays between a service and the corresponding firewall. When a service requesting message arrives at SigFree, SigFree uses a novel technique called linear Genetic programming. The aim of Genetic programming (GP) methodology is discovering rules suitably generic for describing a wide range of anomalous behaviours. However, there are at least two pragmatic limitations constraining the applicability of GP based detectors. Firstly, the datasets used to characterize intrusion detection problems typically consist of millions of exemplars, which implies an overhead in training time. Secondly, once trained, the model is only as good as the data available at training, a third party is again required to provide appropriate labels for new attack instances. The Solutions to the problem have been demonstrated by way of active learning algorithms instructions in an instruction sequence, and then compares the number of useful instructions.

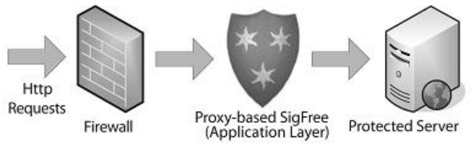

Fig. 1. SigFree is an application layer blocker between the protected server and the corresponding firewall[11].

II. Related Work

Recently, several researchers proposed different detection and prevention methods to detect and prevent the buffer overflow attacks. We have classified these into three categories: 1) Prevention/Detection techniques of Buffer Overflows;2) Worm detection and signature generation;3) Machine code analysis for security purposes.

2.1. Prevention/ Detection of Buffer Overflows

The Existing prevention/detection techniques of buffer over-flows can be roughly broken down into six classes: Class 1A: Finding bugs in source code. Buffer overflows are fundamentally due to programming bugs. Accordingly, various bug-finding tools [13], [14], [15] have been developed. The bug-finding techniques used in these tools, which is general belong to static analysis, include but are not limited to checking and bugs-as-deviant-behaviour. Class 1A techniques are designed to handle source code only, and they do not ensure completeness in bug finding. In contrast, SigFree handles machine code embedded in a request (message). The Class 1B: Compiler extensions.

"If the source code is available, a developer can add buffer overflow detection automatically to a program by using a modified compiler". Three such compilers are StackGuard [16], ProPolice 17], and Return Address Defender (RAD) [18]. DIRA [19] is another compiler that can detect control hijacking attacks, identify the malicious input, and repair the compromised program. Class 1B techniques require the availability of source code. In contrast, SigFree does not need to know any source code. Class 1C: OS modifications. Modifying some aspects of the operating system may prevent buffer overflows such as Pax [20], LibSafe [21], and e-NeXsh [22].

Class 1C: Techniques need to modify the OS. In contrast, SigFree does not need any modification of the OS.

Class 1D: Hardware modifications. A main idea of hard-ware modification is to store all return addresses on the processor [29]. In this way, no input can change any return address.

Class 1E: Defence-side obfuscation. Address Space Layout Randomization (ASLR) is a main component of Pax [21].Address-space randomization, in its general form [23], can detect exploitation of all memory errors. Instruction set randomization [2], [3] can detect all code-injection attacks, while SigFree cannot guarantee detecting all injected code. Nevertheless, when these approaches detect an attack, the victim process is typically terminated. "Repeated attacks will require repeated and expensive application restarts, effectively rendering the service unavailable" [6].

Class 1F: Capturing code running symptoms of buffer overflow attacks. Fundamentally, buffer overflows area code running symptom. If such unique symptom s can be precisely captured, all buffer overflows can be detected.

Class 1B, Class 1C, and Class 1E techniques can capture some but not all—of the running symptoms of buffer overflows. For example, accessing non executable stack segments can be captured by OS modifications; the compiler modifications can detect return address rewriting; and process crash is a symptom capture d by defence-side obfuscation. To achieve 100 percent coverage in capturing buffer overflow symptoms, dynamic data flow/taint analysis/program shepherding techniques were proposed in Vigilante [5], Taint Check [4], and [24]. They can detect buffer overflows during runtime.

Covers [6] and [7]. Post crash symptom diagnosis extracts the "signature" after a buffer overflow attack is detected. A more recent system called ARBOR [25] can automatically generate vulnerability-oriented signatures by identifying characteristic features of attacks and using program context. Moreover, A RBOR automatically invokes the recovery actions.

Class 1F techniques can block both the attack requests that injection code and the attack requests that do not contain any code, but they need the signatures to be firstly generated. Moreover, they are either suffer from significant runtime overhead or need special auditing or diagnosis facilities, which are not commonly available in commercial services. In contrast, although SigFree could not block the attack requests that do not contain any code, SigFree is signature free and does not need any changes to real-world services.

2.2 Worm Detection and Signature Generation

Because buffer overflow is a key target of worms when they propagate from one host to another, SigFree is related to worm detection. Based on the nature of worm infection symptoms, worm detection techniques can be broken down into three classes: [26]. [Class 2B] techniques use such local traffic symptoms as content invariance, content prevalence, and address dispersion to generate worm signatures and/or block worms. Some examples of Class 2B techniques are Earlybird [8], Autograph [9], Polygraph [10], Hamsa [27], and Packet Vaccine [28]. [Class 2C] techniques use worm code running symptoms to detect worms. It is not surprising that Class 2C techniques are exactly Class 1F techniques. Some examples of Class 2C techniques are Shield [29], Vigilante [5], and COVERS [6]. [Class 2D] techniques use anomaly detection on packet payload to detect worms and generate signature. Wang and Stolfo [30], [31] first proposed Class 2D techniques called PAYL. PAYL is first trained with normal network flow traffic and then uses some byte-level statistical measures to detect exploit code.

Class 2A techniques are not relevant to SigFree. Class 2C techniques have already been discussed. Class 2D techniques could be evaded by statistically simulating normal traffic [32]. Class 2B techniques rely on signatures, while SigFree is signature free. Class 2B techniques focus on identifying the unique bytes that a worm packet must carry, while SigFree focuses on determining if a packet contains code or not. Exploiting the content invariance property, Class 2B techniques are typically not very resilient to obfuscation. In contrast, SigFree is immunized from most attack-side obfuscation methods.

2.3 Machine Code Analysis for Security Purposes

Although source code analysis has been extensively studied (see Class 1A), in many real-world scenarios, source code is not available and the ability to analyse binaries is desired. Machine code analysis has three main security purposes: (P1) malware detection, (P2) to analyse obfuscated binaries, and (P3) to identify and analyse the code contained in buffer overflow attack packets.

Along purpose of P1, Chritodorescu and Jha [33] proposed static analysis techniques to detect malicious patterns in executables, and Chritodorescu et al. [34] exploited semantic heuristics to detect obfuscated malware. Along purpose P2, Lakhotia and Eric [35] used static analysis techniques to detect obfuscated calls in binaries, and Kruegel et al. [36] investigated disassembly of obfuscated binaries.

SigFree differs from P1 and P2 techniques in design goals. The purpose of SigFree is to see if a message contains code or not, not to determine if a piece of code has malicious intent or not. Hence, SigFree is immunized from most attack-side obfuscation methods. Nevertheless, both the techniques in [37] and SigFree disassemble binary code, although their disassembly procedures are different. As will be seen, disassembly is not the kernel contribution of SigFree. Fnord [38], the pre-processor of Snort IDS, identifies exploit code by detecting NOP sled. Binary disassembly is also used to find the sequence of execution instructions as an evidence of an NOP sled [12]. However, some attacks such as worm CodeRed do not include NOP sled and, as mentioned in [12], mere binary disassembly is not adequate.

Very recently, Wang etal. [11] proposed a SigFree, an online buffer overflow attack blocker, to protect Internet services. The idea of SigFree is motivated by an important observation that “the nature of communication to and from network services is predominantly or exclusively data and not executable code”. However, their method has following limitations: First, Sig Free cannot fully handle the branch-function-based obfuscation, causes control to be transferred to the corresponding location f(x). By replacing unconditional branches in a program with calls to the branch function, attackers can obscure the flow of control in the program. We note that there are no general solutions for handling branch function at the present state of the art. Second, Sig Free cannot fully handle self-modifying code. Self-modifying code is a piece of code that dynamically modifies itself at runtime and could make Sig Free mistakenly exclude all its instruction sequences. Third, the executable shell codes could be written in alphanumeric form. Such shell codes will be treated as printable ASCII data and thus bypass our analyser. Their Scheme can successfully detect alphanumeric shell codes; however, it will increase computational overhead. Therefore, it requires slight tradeoffs between tight security and system performance.

Fourth, Sig Free does not detect attacks such as return-to-libc attacks that just corrupt control flow or data without injecting code. However, these attacks can be handled by some simple methods.

III. Buffer Overflow Attacks

The core behaviour of an overflow attack lies in the simple observation that just because an address space of a variable declared in a program might be allocated of a specific size, this does not stop the same program from attempting to access memory outside of the allocated space. In order to make use of such a weakness, the attacker requires three components: (1) program used by the target system that possesses inherent overflow vulnerability; (2) Knowledge of the size of memory reference necessary to cause the overflow; and (3) The correct placement of a suitable exploit to make use of the overflow when it occurs. The skill in crafting such an attack lies in how an exploit is hidden and ensuring that the memory referenced outside of the allocated space corresponds to the code defining the desired malicious behaviour.

There are two variants of buffer overflow attacks: 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-163 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 of 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. The 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.

IV. Proposed Method

The proposed work consists of prevention and detection of buffer overflows. This work proposes SigFree, a real-time, signature-free, out-of the-box, application layer blocker for preventing and detecting buffer overflow attacks, which is one of the most serious cyber security threats. The SigFree can filter out code-injection buffer overflow attack messages targeting at various Internet services such as web service. Motivated by the observation that buffer overflow attacks typically contain executables whereas legitimate user requests never contain executables in most Internet services, SigFree blocks attacks by detecting the presence of code.

Our SigFree method using GP first blindly dissembles and extracts instruction sequences from a request. Then, it compares the number of useful instructions to a threshold to determine if this instruction sequence contains code. SigFree is signature free, thus it can block new and unknown buffer overflow attacks; SigFree is also immunized from most attack-side code obfuscation methods. Since SigFree is transparent to the servers being protected, it is good for economical Internet wide deployment with very low deployment and maintenance cost.

The Observations providing the basis for this approach include (1) recognizing that the GP code bloat phenomena provides the basis for the non operational command sequences, thus masking the operational or real intent of an attack command sequence. (2) The SigFree based GP solutions have been widely observed to have functionality distributed across the length of the individual. Thus, the ideal objective of an attack agent will be to build command sequences, which have the same function as the original generic attack, however camouflaged in non-operational but syntactically correct commands. Such a system is only possible if a sufficiently informative fitness function can be defined for the class of attacks as a whole. That is to say, a binary fitness function in which all unsuccessful attacks provide a fitness of zero and a successful attack a fitness of unity, is on the face of it, not much use.

4.1 Basic Fitness Function

Categorically, the attack we are evolving is an 'execve' attack. Execve is a system call in OS that executes a program, where the program takes the form of an argument (UNIX shell /bin/sh in our case). UNIX defines 'execve' as, int execve (const char *path, char *const argv [12], char *const envp [12]). Where the parameter one is the command name; parameter two contains pointers to strings that will be given to the program as arguments; parameter three contains pointers to environmental variables, which are also stored as strings. A minimalist call to the 'execve' function using the C language might have the following form in algorithm 1[12]:

Algorithm 1: Minimal List

```
int main()
{
  char *command = "/bin/sh";
  char *args[2];
  args[0] = command;
  args[1] = 0;
  execve(command, args, 0);
}
```

In order to spawn a UNIX shell prompt, 'execve' requires that the command pointer should be in the EBX register, the pointer to 'args' should be in register ECX and the pointer to the third argument (which is in our case is the NULL pointer) should be in

EDX register, Algorithm 2[11]. Moreover, the program name '/bin/sh' should be pushed to stack. To achieve these goals, 11 assembly instructions are needed. After 10 instructions are executed (11th is the interrupt that transfers control to execve system call), registers EAX, EBX, ECX and EDX should be correctly configured and the stack should contain the program name to be executed (i.e. 1668 /bin/sh). Given the state of the stack and registers after the 10th instruction, if the values are not set correctly, greedy replacement is used to determine how many instructions are needed to correct it. For example if "/bin/sh" has not been pushed to the stack, 3 instructions are sufficient to achieve this goal. Another important point is that if the task has been half-accomplished, viable instructions should be determined. Using this principle, the fitness function summarized in Algorithm 3[12] returns a maximum fitness of 10 if all conditions are satisfied; otherwise it subtracts the number of instructions needed to correct the program, relative to the minimal set of sub-goals, Algorithm 2. The basic fitness function therefore it takes the form of a hierarchical fitness function in which sub-goals (a) to (e) can only be completed in sequence. However, depending on the composition of the language used to evolve the attacks, there are multiple programs producing the required (buffer overflow) attack behaviour. The algorithm 1 describes about this:

Algorithm 2: Minimal requirements for executing an execve' system call for spawning a UNIX shell.

1. Register EAX contains 0x0B i.e., the system call number of 'execve';
2. Register EBX points to '/bin/sh0' on the stack;
3. Register ECX points to the argument array in stack;
4. Register EDX contains NULL;
5. Interrupt '0x80' is executed;

Algorithm 3: Basic fitness functions for establishing correct behavior of 'execve' exploit. Fitness = 10

(a) IF stack does not contain '/bin/sh0', THEN subtract number of instructions necessary to do so from Fitness (1 to 3).
(b) IF register EBX does not point to string from (a), THEN Fitness =1;
(c) IF register ECX does not point to argument array in stack, THEN subtract number of instructions necessary to do so from Fitness (1 to 3)
(d) IF register EDX! = NULL, THEN Fitness=1.
(e) IF an INT is not executed, THEN Fitness.

4.2. Linear GP

Individual users are represented using linear GP in which instructions are composed from a 2-byte opcode and two operands (each 1-byte) i.e. all instructions have the same number of bytes. The Individual users are defined using a fixed length format, thus initialization is defined in table 1 over the total range of permitted program lengths. Selection takes the form of a steady state tournament over 4 individuals. The children from the best performing half of the tournament overwrite the individuals corresponding to the worst half of the tournament, taking their place in the population. Search operators take three forms: two point crossover, instruction mutation, and instruction swap. Therefore, constrained to exchange an equal number of instructions (a page) between two individuals. The number of instructions per page is allowed to vary from 1 instruction to max instructions per page as the fitness function reaches a new plateau, as in the page-based Linear GP framework [12]. Mutation selects a single instruction with uniform probability and replaces with a different instruction from the instruction set, Table 1[12]. The swap operator selects two instructions from the same individual with equal probability and interchanges their respective positions.

The details of the linear Genetic Programming methodology itself is not particularly important, however, previous work utilizing Code Abstraction[11] indicated that the linear representation provides a more direct method for successfully evolving buffer overflow attacks (the search operators in GE were not particularly efficient at manipulating register references) [12].

Table 1
GP Parameters[12]

Parameter	Setting
Crossover	Page based crossover with 0.9 probability
Mutation	Uniform instruction-wide mutation. with 0.5 probability
Swap	Instruction swap within an individual with 0.5 probability
Selection	Tournament of 4 individuals
Stop Criteria	At the end of 50,000 tournaments.
Population	500 individuals with 10 pages and 3 instructions per page.
Training Time	Approximately 6 hours.

V. SECURITY ANALYSIS

In this section, we analyse the security of our scheme as follows: As indicated in the introduction, two basic approaches are considered in the design of buffer overflow attacks. Our Objective is to develop the attack itself whilst maximizing the probability of executing the malicious code, CA [11] proved very inefficient at manipulating register references (using the standard CA search operators) than Sig-free method using code abstraction. By using linearly structured GP, we expect to avoid this problem. In the following we describe a series of three experiments in which the instruction set is incrementally expanded, thus increasing the search space, but providing for greater freedom in the resulting program content (thus a wider range of behavioural properties). This case results in code that has the capacity to intermix attack and obfuscation.

In all cases the fitness function takes the form of Algorithm 2, augmented with an additional term to measure the likelihood of an attack being executed. Specifically, since all individuals have a fixed length of 30 instructions and it takes 11 instructions to describe the attack, there are up to 19 instructions denoting introns with respect to the malicious code. If the approximated return address was not accurate enough to jump to the first instruction, jumping to an effective useless region would allow an attack to deploy successfully. If execution of a successful attack fails (i.e. an inaccurate return address) by jumping past a relevant instruction, the location of the instruction is called the failure point. The probability of execution is defined as (failure point ÷ number of all possible points); or a denominator of 19 in this case.

Compared to existing scheme [11], our scheme based Genetic Programming blocks the all buffer overflow attacks.

Therefore, SigFree Blocker using Genetic Programming is securing than SigFree using code abstraction while detecting the attacks.

VI. PERFORMANCE ANALYSIS

In this section, we implemented a Sig-Free prototype using the C programming language in the Win64 environment. The stand-alone prototype was com-piled with Borland C++ version 5.5.1 at optimization level O2. The experiments were performed in a Windows 2007 server with Intel Pentium 4, 3.2-GHz CPU, and 1-Gbyte memory. We measured the processing time of the stand-alone prototype over all (2,910 totally) 0-10 Kbyte images collected from the above real traces. We set the upper limit to 10 Kbytes because the size of a normal web request is rarely over that if it accepts binary inputs.

To evaluate the performance impact of SigFree to web servers, we also implemented a proxy-based SigFree prototype. Fig. 2 depicts the implementation architecture. It is comprised of the following modules. URI decoder. The specification for URLs limits the allowed characters in a Request-URI to only a subset of the ASCII character set. This means that the query parameters of a request-URI beyond this subset should be encoded [39].

Because a malicious payload may be embedded in the request-URI as a request parameter, the first step of SigFree is to decode the request-URI. ASCII filters. Malicious executable codes are normally binary strings. In order to guarantee the throughput and response time of the protected web system, if a request is printable ASCII ranging from 20 to 7E in hex, SigFree allows the request to pass.

The proxy-based prototype was also compiled with Borland C++ version 5.5.1 at optimization level O2. The proxy-based prototype implementation was hosted in the Windows 2007 server with Intel Pentium 4, 3.2-GHz CPU, and 1-Gbyte memory. The proxy-based SigFree prototype accepts and analyses all incoming requests from clients. The client testing traffics were generated by JefPoskanzer's http_load program [40] from a Linux desktop PC with Intel Pentium 4 and 2.5-GHz CPU connected to the Windows server via a 100-Mbps LAN switch. We modified the original http_ load program so that clients can send code-injected data requests.

For the requests that SigFree identifies as normal, SigFree forwards them to the web server, Apache HTTP Server 2.0.54 hosted in a Linux server with dual Intel Xeon 1.8-Gbyte CPUs. The individual Clients send requests from a predefined URL list. The documents referred in the URL list are stored in the web server.

In addition, the prototype implementation uses a time-to-live-based cache to reduce redundant HTTP connections and data transfers. Rather than testing the absolute performance overhead of SigFree, we consider it more meaningful measuring the impact of SigFree on the normal web services. Hence, we measured the average response latency of the connections by running http load for 1,000 fetches. Whenever there are no buffer overflow attacks, the average response time in the system with SigFree is only slightly higher than the system without SigFree. This indicates that, despite the connection and ASCII checking overheads, the proxy-based implementations does not affect the overall latency significantly.

The Fig.2 shows the average latency of connections as a function of the percentage of attacking traffic. We used CodeRed as the attacking data. Only successful connections were used to calculate the average latency; that is, the latencies of attacking connections were not counted. This is because what we care is the impact of attack requests on normal requests. We observe that the average latency increases slightly worse than linear when the percentage of malicious attacks increases. Generally, proposes scheme using Genetic Programming [12] is about 40 percent faster than SigFree using code abstraction [11] and existing scheme[11] is slightly slower than our scheme.

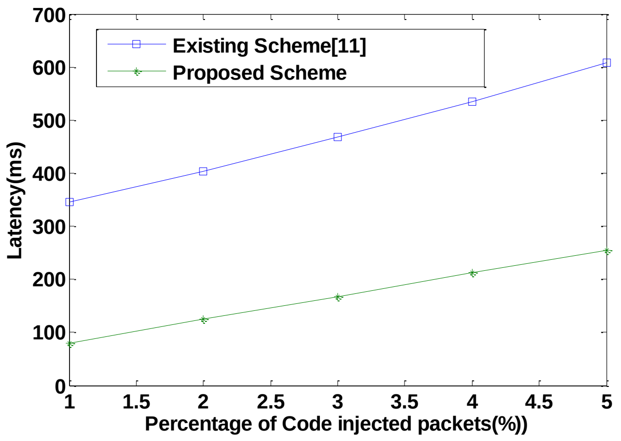

Fig.2: Performance impact of SigFree in Proposed and Existing schemes on HTTP server

VII. Conclusion

We have proposed A novel Signature-Free Buffer Overflow Attack Blocker Using Genetic Programming that can filter code-injection buffer overflow attack messages, one of the most serious cyber security threats. Our method does not require any signatures, thus it can block new unknown attacks.

SigFree is immunized from most attack-side and good for economical Internet-wide deployment with little maintenance cost and low performance overhead. The Results show that code bloat property of the GP provides suitable means to hide the actual attack by mixing exploit instructions with introns that have no effect toward the success of the attack. Furthermore, evolved attacks discover different ways of attaining sub-goals associated with building buffer overflow attacks, hence mimicking the core attack with different instructions. Finally, we have proved that our method is secured than existing scheme [11].

REFERENCES

[1] B.A. Kuperman, C.E. Brodley, H. Ozdoganoglu, T.N. Vijaykumar, and A. Jalote, “Detecting and Prevention of Stack Buffer Overflow Attacks,” Comm. ACM, vol. 48, no. 11, 2005.

[2] G. Kc, A. Keromytis, and V. Prevelakis, “Countering Code-Injection Attacks with Instruction-Set Randomization,” Proc. 10th ACM Conf. Computer and Comm. Security (CCS ’03), Oct. 2003.

[3] E. Barrantes, D. Ackley, T. Palmer, D. Stefanovic, and D. Zovi, “Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks,” Proc. 10th ACM Conf. Computer and Comm. Security (CCS ’03), Oct. 2003.

[4] J. Newsome and D. Song, “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software,” Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS), 2005.

[5] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, “Vigilante: End-to-End Containment of Internet Worms,” Proc. 20th ACM Symp. Operating Systems Principles (SOSP), 2005.

[6] Z. Liang and R. Sekar, “Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers,” Proc.12th ACM Conf. Computer and Comm. Security (CCS), 2005.

[7] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt, “Automatic Diagnosis and Response to Memory Corruption Vulnerabilities,” Proc. 12th ACM Conf. Computer and Comm. Security (CCS), 2005.

[8] S. Singh, C. Estan, G. Varghese, and S. Savage, “The Early bird System for Real-Time Detection of Unknown Worms,” technical report, Univ. of California, San Diego, 2003.

[9] H.-A.Kim and B. Karp, “Autograph: Toward Automated, Distributed Worm Signature Detection,” Proc. 1 3t h U SE NI X Security Symp. (Security), 2004.

[10] J. Newsome, B. Karp, and D. Song, “Polygraph: Automatic Signature Generation for Polymorphic Worms,” Proc. IEEE Symp.Security and Privacy (S&P), 2005.

[11] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Z hu, “SigFree: A Signature-Free Buffer Overflow Attack Blocker”, IEEE TRANSAC TIONS ON DEPEN DABLE AND SECURE COMPUTING, VOL. 7, NO. 1, JANUAR Y-MARCH 2010.

[12] Hilmi Güneş Kayacık, Malcolm Heywood, Nur Zincir-Heywood, “On Evolving Buffer Overflow Attacks Using Genetic Programming”, In Proc. Of GECCO’06, July 8–12, 2006, Seattle, Washington, USA.

[13] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken, "A First Step towards Automated Detection of Buffer Overrun Vulnerabilities,"Proc. Seventh Ann. Network and Distributed System Security Symp. (NDSS '00), Feb. 2000.

[14] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," IEEE Software, vol. 19, no. 1, 2002.

[15] H. Chen, D. Dean, and D. Wagner, "Model Checking One Million Lines of C Code," Proc. 11th Ann. Network and Distributed System Security Symp. (NDSS), 2004.

[16] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "Stackguard:Automatic Adaptive Detection an d Prevent ion of Buffer-Overflow Attacks, " Proc. Se v en t h US EN I X Security Sym p .(Security '98), Jan. 1998.

[17] GCC Extension for Protecting Applications from Stack - S mashing Attacks, http:/ /www .research.Ibm.com/trl/projects/security/ssp, 2007.

[18] T. cker Chiueh and F.-H. Hsu, "Rad: A Compile-Time Solution to Buffer Overflow Attacks," Proc. 21st Int'l Conf. Distributed Computing Systems (ICDCS), 2001.

[19] A. Smirnov and T.cker Chiueh, "Dira: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks," Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS), 2005.

[20] Pax Documentation, http://pax.grsecurity.net/docs/pa x.txt, Nov. 2003.

[21] A. Baratloo, N. Sing h, and T. Tsai, "Trans parent Run-Time defense against Stack Smashing Attacks," Proc. USENIX Ann. Technical Conf. (USENIX '00), June 2000.

[22] G.S.Kc and A.D.Keromytis, "E-NEXSH: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing," Proc.21st Ann. Computer Security Applications Conf. (ACSAC), 2005.

[23] S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," Proc.14th USENIX Security Symp. (Security), 2005.

[24] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure Execution via Program Shepherding," Proc. 11th USENIX Security Symp. (Security), 2002

[25] Z. Liang and R. Sekar, "Automatic Generation of Buffer Overflow Attack Signatures: An Approach Based on Program Behavior Models," Proc. 21st Ann. Computer Security Applications Conf. (ACSAC), 2005.

[26] R.Pang, V.Yegneswaran, P.Barford, V.Paxson, and L. Peterson, "Characteristics of Internet Background Radiation," Proc. ACM Internet Measurement Conf. (IMC), 2004.

[27] Z. Li, M. Sanghi, Y. Chen, M.Y. Kao, and B. Chavez, "Hamsa: Fast Signature Generation for Zero-Day Polymorphic Worms with Provable Attack Resilience," Proc. IEEE Symp. Security and Privacy (S&P '06), May 2006.

[28] X.F. Wang, Z. Li, J. Xu, M.K. Reiter, C. Kil, and J.Y. Choi, "Packet Vaccine: Black-Box Exploit Detection and Signature Generation,"Proc. 13th ACM Conf. Computer and Comm. Security (CCS), 2006.

[29] H.J.Wang, C.Guo, D.R.Simon, and A.Zugenmaier, "Shield:

Vulnerability - Driven Network Filters for Preventing Known Vulnerability Exploits," Proc. ACM SIGCOMM '04, Aug. 2004.

[30] K. Wang and S.J. Stolfo, "Anomalous Payload-Based Network Intrusion Detection," Proc. Seventh Int'l Symp. Recent Advances in Intrusion Detection (RAID), 2004.

[31] K. Wang, G. Cretu, and S.J. Stolfo, "Anomalous Payload-Based Worm Detection and Signature Generation," Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID), 2005.

[32] O. Kolesnikov, D. Dagon, and W. Lee, "Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic," Technical Report GIT-CC-04-13, College of Computing, Georgia Tech, 2004.

[33] M. Christodorescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," Proc. 12th USENIX Security Symp. (Security '03), Aug. 2003.

[34] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant, "Semantics-Aware Malware Detection," Proc. IEEE Symp. Security and Privacy (S&P), 2005.

[35] A.Lakhotia and U. Eric, "Abstract Stack Graph to Detect Obfuscated Calls in Binaries," Proc. Fourth IEEE Int'l Workshop Source Code Analysis and Manipulation (SCAM '04), Sept. 2004.

[36] C. Kruegel, W. Robertson, F.Valeur, and G.Vigna, "Static Disassembly of Obfuscated Binaries," Proc. 13th USENIX Security Symp. (Security), 2004.

[37] Fnord Snort Preprocessor, http://www.c ansecwest.com/spp_fnord.c, 2007.

[38] B. Schwarz, S.K. Debray, and G.R. Andrews, "Disassembly of Executable Code Revisited," Proc. Ninth IEEE Working Conf.Reverse Eng. (WCRE), 2002.

[39] T. Berners-Lee, L.Masinter, and M. McCahill, Uniform Resource Locators (URL), RF C 17 38 (Proposed Standard), updated by RFCs 1808, 2368, 2396, 3986, http://www.ietf.org/rfc/rfc1738.txt,2007.

[40] Http Load: Multiprocessing Http Test Client, http://www.acme.com/software/http_load, 2007.