ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization) Vol.2, Special Issue 1, March 2014

Proceedings of International Conference On Global Innovations In Computing Technology (ICGICT'14)

Organized by

Department of CSE, JayShriram Group of Institutions, Tirupur, Tamilnadu, India on 6th & 7th March 2014

Taint Flow Analysis for the Detection of Buffer Overflow Attacks

Abu Imran K[1]

II M.E .CSE, Park College of Engineering and Technology, Coimbatore, Inida[1]

Abstract: The buffer overflow defense employs generic code-data separation criteria on the disassembled payloads to distinguish between code embedded payloads and data payloads. Static analysis based detection mechanisms allow detection of new or previously unknown attacks .The static taint analysis coupled with implicit taint flow analysis improve the detection effectiveness of malcode detector.

Index Terms: buffer overflow, code injection attack, code obfuscation, self modifying code, implicit flow analysis, static taint analysis

I. INTRODUCTION

THE buffer overflow remains the crown jewel of attacks, and it is likely to remain so for years to come. A buffer overflow occurs during program execution when a fixed-size buffer has had too much data copied into it. The buffer overflow vulnerability is the most common injection vector of code injection attacks. The code injection attack is a type of attack where an attacker sends executable code to a vulnerable host that is executed on behalf of the attacker .If such an attack is possible the security of the entire system is compromised and the attacker is able to perform arbitrary operations on the compromised host.

The existing defenses may cause substantial changes to existing (legacy) server OSes, application software, and hardware, thus they are not transparent [4].Also the IDS that is based on signatures fail to detect new or previously unknown buffer overflow attacks.

In order to detect the buffer overflow attacks effectively a signature free detection mechanism is used. The idea proposed is that if the generic code data separation criterion is used, it would help protect those web applications whose nature of communication is predominantly and exclusively data and not executable code.

However, the generic code- data separation criteria is not effective in detecting self modifying code. The static taint analysis and implicit taint flow analysis are integrated to the signature free defense mechanism to allow the detection of self modifying code. iism

The rest of this paper is organized as follows: In Section 2, the related work is summarized. In Section 3, the implementation of proxy based IDS is discussed. In Section 4, experimental results are shown. The paper is concluded in section 5.

II. RELATED WORK

2.1. POLYMORPHIC WORM DETECTION USING STRUCTURAL NFORMATION OF EXECUTABLES

The worms can be polymorphic; that is, they can mutate as they spread across the network. To detect these types of worms, it is necessary to devise new techniques that are able to identify similarities between different mutations of a

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization) Vol.2, Special Issue 1, March 2014

Proceedings of International Conference On Global Innovations In Computing Technology (ICGICT'14)

Organized by

Department of CSE, JayShriram Group of Institutions, Tirupur, Tamilnadu, India on 6th & 7th March 2014

worm. The structural analysis of binary code that allows one to identify structural similarities between different worm mutations [1].

The approach is based on the analysis of a worm's control flow graph and introduces an original graph coloring technique that supports a more precise characterization of the worm's structure. The analysis is complex, and, thus, more costly since it needs to parse the network stream into instructions, building the control flow graph, generate subgraphs, and perform canonical graph labelling .Also, an attacker can easily evade detection by producing structurally similar executables with instructions that result in different colorings.

2.2. ANALYZING NETWORK TRAFFIC TO DETECT SELF-DECRYPTING EXPLOIT CODE

The overall idea [2] is to scan the network traffic for the presence of the decryption routine which is characteristic of polymorphic exploit code. Static analysis is used to locate the decryption routine inside the network traffic. Limited emulation of instruction execution is performed to reveal concealed components such as self-modifying instructions of the decryption routine. The method works by scanning the network traffic for the presence of a decryption routine, which is characteristic of such exploits [2]. First, it identifies the possible starting locations of a decryption routine by looking for a form of GetPC code.

Second, it finds the actual decryption instructions by a novel two-way traversal of the code, as well as by using standard backward data flow analysis. Third, it identifies self-modifying decryption routines through emulated execution of already found decryption instructions. Last, it verifies that the detected code is a decryption routine by checking whether it satisfies two properties that are typical of such code.

The first property is that in a detected loop, there must be a memory-write instruction that uses indirect addressing. The second property is that the register holding the address or offset must be updated within the loop. Otherwise the same memory location will be written over and over. The disadvantages of this approach are fragmentation, no use of looping by the decryption routine, use of values not in the exploit code and long or infinite loops.

2.3.A FAST STATIC ANALYSIS APPROACH TO DETECT EXPLOIT CODE INSIDE NETWORK FLOWS

The approach to exploit detection is to look for evidence of meaningful data and control flow, essentially focusing on both NOOP sled and payload components whenever possible [3]. An important consequence of using a static analysis based approach is that it can not only detect previously unseen exploit code but is also more resilient to changes in implementation which exploit code authors employ to defeat signature-based techniques.

Having performed binary disassembly using convergent binary disassembly strategies the control flow graph (CFG) is constructed. Each node of the CFG signifies the basic block. The three states with each basic block - valid, if the branch instruction at the end of the block has a valid branch target, invalid, if the branch target is invalid, and unknown, if the branch target is not obvious, are identified. This information helps in pruning the CFG.

The data flow analysis based on program slicing is used to complete the process of elimination. Program slicing is a decomposition technique which extracts only parts of a program relevant to a specific computation. For the last block in a chain, the following cases capture the nature of the branch instruction.

Case I: Obvious Library Call.
Case II: Obvious Interrupt.
Case III: The ret Instruction.
Case IV: Hidden Branch Target.

The disadvantages of this approach are

- More overhead and longer latency in filtering packets due to more rules.
- Rules are initial sets and may require updating over time, so the attackers can bypass those preknown rules.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization) Vol.2, Special Issue 1, March 2014

Proceedings of International Conference On Global Innovations In Computing Technology (ICGICT'14)

Organized by

Department of CSE, JayShriram Group of Institutions, Tirupur, Tamilnadu, India on 6th & 7th March 2014

2.4. SIGFREE: A SIGNATURE-FREE BUFFER OVERFLOW ATTACK BLOCKER

SigFree, an online signature-free out-of-the-box application-layer method 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 client requests never contain executables in most Internet service, SigFree blocks attacks by detecting the presence of code [4].

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". When a service requesting message arrives at SigFree, SigFree first disassembles and distill all possible instruction sequences from the message's payload. However, in this phase, some data bytes may be mistakenly decoded as instructions. In phase 2 SigFree uses a novel technique called code abstraction. Code abstraction first uses data flow anomaly to prune useless instructions in an instruction sequence, then compares the number of useful instructions (Scheme 2) or dependence degree (Scheme 3) to a threshold to determine if this instruction sequence (distilled in phase 1) contains code. Unlike the existing code detection algorithms that are based on signatures, rules, or control flow detection, SigFree is generic and hard for exploit code to evade.

The disadvantages of this approach are:

- SigFree cannot fully handle the branch-function-based obfuscation.
- SigFree cannot fully handle self-modifying code.
- SigFree does not detect attacks such as return-to libc attacks.
- ASCII filter cannot detect the executable shellcodes that could be written in alphanumeric form.

2.5. STILL: EXPLOIT CODE DETECTION VIA STATIC TAINT AND INITIALIZATION ANALYSES

STILL, a novel Static Taint and InitiaLization anaLysis based approach [5] to detect not only unobfuscated exploit code (without any obfuscation), traditional polymorphic and metamorphic exploit code, but also self-modifying and indirect jump obfuscation code. STILL is based on the same observation as in that remote exploits typically contain executables, whereas legitimate client requests never contain executables in most Internet services [5].

STILL detects attacks as follows. It works as a proxy-based blocker in the application layer of clients and/or servers. When it captures a data stream, it disassembles the data stream and generates a control flow graph. It analyzes the disassembled result in two stages. First, STILL detects self-modifying and indirect jump obfuscation code. Although the real exploit code may be hidden by self-modifying and indirect jump, the obfuscation code itself provides some strong evidence of self-modifying and/or indirect jump behaviour. STILL detects this behaviour by static taint analysis and initialization analysis.

The disadvantages of this approach are:

- STILL does not handle memory address obfuscation.
- STILL does not handle implicit flow in static taint analysis.
- STILL does not detect return-to-libc attacks which do not contain any code.

III. ENHANCED TAINT FLOW ANALYSIS FOR THE DETECTION OF BUFFER OVERFLOW ATTACKS

The signature free defense mechanism relies on the generic code–data separation criteria. The static analysis reduces the runtime overhead when compared with dynamic analyses. The focus is on those attack packets whose payload contains executable code in machine language and it is assumed that normal packets do not contain executable machine code.

The IDS is implemented as an application layer blocker that typically stays between a service and the corresponding firewall. When a service requesting message arrives at the IDS, it uses the recursive traversal disassembly algorithm to disassemble and distill all possible instruction sequences from the message's payload. However, in this phase, some data

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization) Vol.2, Special Issue 1, March 2014

Proceedings of International Conference On Global Innovations In Computing Technology (ICGICT'14)

Organized by

Department of CSE, JayShriram Group of Institutions, Tirupur, Tamilnadu, India on 6th & 7th March 2014

bytes may be mistakenly decoded as instructions. Therefore in the next phase data flow anomalies are used to differentiate between random program sequences and program fragments. Code abstraction uses data flow anomaly to prune useless instructions in an instruction sequence, then compares the number of useful instructions or dependence degree to a threshold to determine if this instruction sequence (distilled in phase 1) contains code.To detect self-modifying and indirect jump obfuscation code static taint analysis and initialization analysis is used. First, the variable which holds the absolute address of the payload is found in the instruction sequences and used as a taint seed. Then, static taint analysis is used to track the tainted values and detect whether tainted data are used in the abstract semantics that could indicate the presence of self-modifying and indirect jump exploit code. The initialization analysis is used to reduce false positives.The implicit flow of the tainted variables are also analysed.

Implementation

The IDS is implemented as an application level proxy as shown in Fig 1 .It is composed of six modules namely URL Decoder, ASCII Filter, Instruction Distiller , Instruction Analyser ,Static Taint and Initialization Analyzer ,Implicit flow Analyzer.

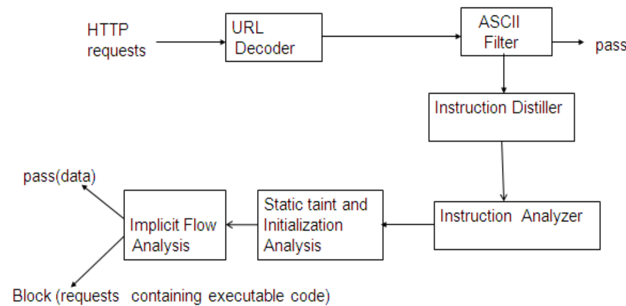

Fig 1.Implementation of proxy based IDS

3.1. URL Decoder

The specification for URLs [6] 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 .Because a malicious payload may be embedded in the request-URI as a request parameter, the first step is to decode the request-URI.

3.2. ASCII Filter

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, the request is allowed to pass. The ASCII filter does not prevent the service from receiving non-ASCII strings. All non-ASCII strings will be analyzed by ISD and ISA.

3.3. Packet Disassembly and Instruction Sequence Distiller

The disassembly is done using recursive traversal disassembly algorithm .Every byte of the request is first assigned with an address (starting from zero) .Then, the request is disassembled from a certain address and the control flow of instruction is followed until the end of the request is reached or an illegal instruction opcode is encountered. Fig. 2 shows four instruction sequences distilled from a substring of a GIF file. Each instruction sequence is denoted as s_i , where i is the entry location of the instruction sequence in the string. An address is assigned to every byte of the string. Instruction sequences s00, s01, s02, and s08 are distilled by disassembling the string from addresses 00, 01, 02, and 08, respectively.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization) Vol.2, Special Issue 1, March 2014

Proceedings of International Conference On Global Innovations In Computing Technology (ICGICT'14)

Organized by

Department of CSE, JayShriram Group of Institutions, Tirupur, Tamilnadu, India on 6th & 7th March 2014

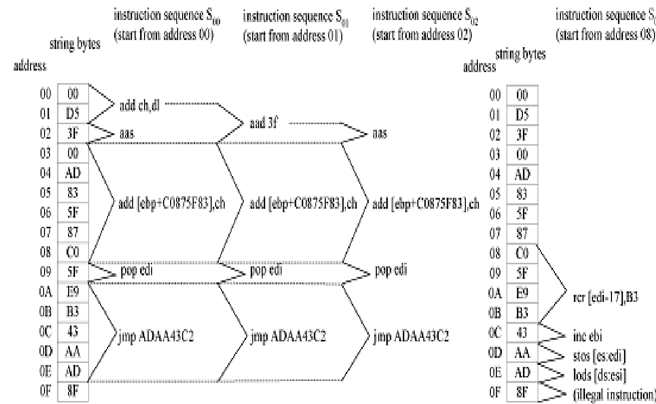

Fig.2. Instruction sequences distilled from a substring of a GIF file.

After disassembling the possible instruction sequences an EIFG for the request is created. An extended IFG (EIFG) is a directed graph G=(V,E),which satisfies the following properties: each node v ε V corresponds to an instruction, an illegal instruction (an "instruction" that cannot be recognized by CPU), or an external address (a location that is beyond the address scope of all instructions in this graph);each edge eεE corresponds to a possible transfer of control from instruction vi to instruction vj, to illegal instruction vj, or to an external address vj.Fig.3 shows the EIFG constructed for the instruction sequences in Fig 2.

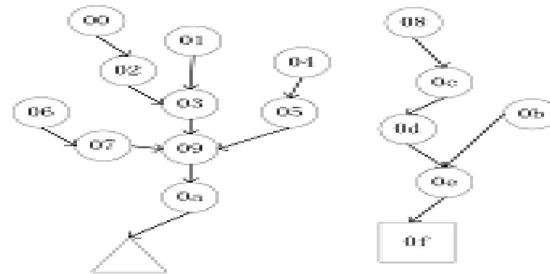

Fig 3.EIFG for the instruction sequence distilled in Fig 2.

The distilling phase may output many instruction sequences at different entry points. Some of these are excluded based on the heuristics. An instruction sequence is excluded if its entry is not the real entry to the embedded code.

Accordingly an instruction sequence s_a is excluded if:

- It is a subsequence of the instruction sequence s_b.
- It merges to the instruction sequence s_b after few instructions.
- Whenever executed an illegal instruction is inevitably reached.

3.4. Instruction Sequence Analyzer

The instruction sequences are analyzed to check whether they are random instruction sequences or real program fragments. The analysis is carried out as three schemes.

Scheme 1: A program has certain characteristics implying the operating system on which it is running, for example calls to operating system or kernel library. A random instruction sequence does not carry this kind of characteristics. Instructions such as "call" and "int 0x2eh" in Windows and "int 0x80h" in Linux may indicate system calls or function

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering
(An ISO 3297: 2007 Certified Organization) Vol.2, Special Issue 1, March 2014
Proceedings of International Conference On Global Innovations In Computing Technology (ICGICT'14)
Organized by
Department of CSE, JayShriram Group of Institutions, Tirupur, Tamilnadu, India on 6th & 7th March 2014

calls. Before these call instructions there are normally one or several instructions used to transfer parameters. For example, a "push" instruction is used to transfer parameters for a "call" instruction; some instructions that set values to registers al, ah, ax, or eax are used to transfer parameters for "int" instructions. These call patterns are very common in a fragment of a real program. By identifying the call pattern in an instruction sequence, a real program can be effectively differentiated from a random instruction sequence[4]. But this scheme is vulnerable to obfuscation .The attacker may replace the push-call sequences with other instructions.

Scheme 2: The scheme 2 uses code abstraction to overcome the shortcomings of scheme 1. Code abstraction is a technique used to detect data flow anomalies. As a result of the code abstraction of an instruction, a variable could be in one of the six possible states. The six possible states are state U: undefined; state D: defined but not referenced; state R: defined and referenced; state DD: abnormal state define-define; state UR: abnormal state undefine-reference; and state DU: abnormal state define-undefine[4].

Normally, a random instruction sequence is full of data flow anomalies, whereas a real program has few or no data flow anomalies. When there are data flow anomalies in an execution path of an instruction sequence, some instructions are useless, whereas in a real program at least one execution path has a certain number of useful instructions.

The useless instructions are pruned from the instruction sequence and the numbers of useful instructions are then compared with a threshold. If the number of useful instructions in an execution path exceeds a threshold, the instruction sequence is identified as a segment of a program. The scheme would fail if the attackers use specially crafted code, knowing the threshold.

Scheme 3: In this scheme the dependency degree of every instruction in the instruction sequence is calculated[4]. Dependency is a binary relation over instructions in an instruction sequence. An instruction j is said to depend on instruction i if instruction i produces a result directly or indirectly used by instruction j. Dependency relation is transitive, that is, if i depends on j and j depends on k, then i depend on k. To calculate the dependence degree of an instruction, a def-use graph is constructed[4]. A def-use graph is a directed graph G= (V,E) where each node vεV corresponds to an instruction and each edge e= (vi,vj) ε E indicates that instruction vj produces a result directly used by instruction vi. Obviously, the number of instructions that an instruction can reach through any path in the def-use graph is the dependence degree of the instruction. If the number of useful instructions a useful instruction depends on exceeds a threshold in an instruction sequence, it is concluded that there are real code is embedded in the request.

3.5. Static Taint and Initialization Analysis

The identification of the injected code by the aforementioned method fails in the case of self-modifying code. Self-modifying code is code that modifies itself when being executed. Self-modifying code is a very powerful technique to thwart static analysis since it can completely hide the semantics of original instructions[5].

To detect self modifying code, taint analysis is used. The first step here is to identify the taint seed. The taint seed is the variable that holds the absolute address of the payload .The attackers will not hardcode this value for the fear of detection. The only way to obtain the absolute address during runtime is to read the PC (Program Counter) value.

The two ways to read the PC value are:

- Using a relative call.
- Using the fstenv instruction.

After the taint seed is found, static taint analysis approach is used to statically determine which variables are tainted in an instruction sequence. A taint seed itself is a tainted variable. A tainted variable is propagated to a new tainted variable by data transfer instructions that move data (e.g., push, pop, move) and data operation instructions that perform arithmetic or bit-logic operations on data (e.g., add, sub, xor) in the IA-32 instruction set.

For data operation instructions, the destination operand will be tainted if and only if either the source or the destination operand is tainted. A variable is tainted if it is tainted in any one of the execution paths. The tainted set equation used is:

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering
(An ISO 3297: 2007 Certified Organization) Vol.2, Special Issue 1, March 2014
Proceedings of International Conference On Global Innovations In Computing Technology (ICGICT'14)
Organized by
Department of CSE, JayShriram Group of Institutions, Tirupur, Tamilnadu, India on 6th & 7th March 2014

$T(i) = U_{x\epsilon pred\ (i)}$ (if(USE(x) U T(x)≠ Φ) then T(x) U DEF(x) else T(x) - DEF(x) where i denotes an instruction T(i) is the set of tainted variables from the entry to instruction i, Pred(i) is the set of i's predecessor instructions in the CFG,USE(x) is the set of variables used in instruction x;DEF(x) is the set of variables defined by instruction x.

The two cases where tainted data indicate self-modifying code obfuscation are shown in Fig 4. First, the tainted data are used as the address of the updating instructions. Second, the tainted data are used as the address of a memory read instruction or the address of a memory write instruction. It is noted that the read result will be used to generate the write result; therefore, a new taint analysis process is started to taint the read result .If the newly tainted data are used as the source operand of a memory write instruction, it clearly indicates self-modifying code obfuscation[5].

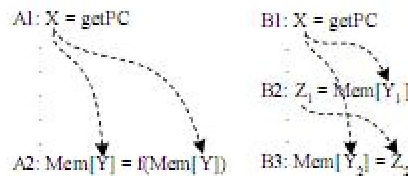

Fig 4. The two cases of tainted data that indicate self-modifying code.

To reduce the false positives, initialization analysis is used. For a self – modifying code it is required that the operands of memory updating or writing instructions are initialized. If these operands are uninitialized, it is not considered as an attack.

There are two special cases of variable initialization. One case is the PC (Program Counter) value, which is always considered to be initialized. The other case is for the instructions whose output is independent of their input. For example, instructions such as "xor eax, eax" or "sub eax, eax" always set eax to zero regardless of the value stored in eax. In this project, these special instructions are recognized and the result variables are considered to be initialized.

An initialized set equation at the entry of an instruction is defined as following:

$I(i) = U_{x\epsilon pred\ (i)}$ (if(USE(x) ⊆ I(x) then I(x) U DEF(x) else I(x) - DEF(x) where i denotes an instruction, I(i) is the set of initialized variables at the entry of instruction i, Pred(i) is the set of i's predecessor instructions in CFG, USE(x) is the set of variables used in instruction x, and DEF(x) is the set of variables defined by instruction x [5].

3.6. Implicit Flow Analysis

Implicit flows signal information through the control structure of a program. For example, a piece of code "for i = 0 to 232 f if (j == i) k = i;" is semantically the same as k = j for 32-bit integer but would not cause k tainted by j in static taint analysis. Therefore, attackers exploiting implicit flow could evade detection. One possible solution is that a variable will be considered tainted if it is modified within the if statements and the condition expression contain a tainted value. This conservative approach will defeat this attack.

IV. RESULTS

The URL Decoder and the ASCII Filter have been implemented and the current work is centered on packet disassembly and EIFG generation.

ISSN(Online): 2320-9801
ISSN (Print): 2320-9798

International Journal of Innovative Research in Computer and Communication Engineering

(An ISO 3297: 2007 Certified Organization) Vol.2, Special Issue 1, March 2014

Proceedings of International Conference On Global Innovations In Computing Technology (ICGICT'14)

Organized by

Department of CSE, JayShriram Group of Institutions, Tirupur, Tamilnadu, India on 6th & 7th March 2014

Fig 5:URL Decoder and ASCII Filter output

Fig 6: Instruction Sequence Distiller output

```
Output - JavaApplication1 (run)    Tasks
push-call count is: 14
reference var is: [0x7C91B077, 0x50, 0x7C91B030, 0x7C90E8CB, EBP-0x28, EBP-0x19, EBP-0x2C, 0x18, EBP-0x30, C
ins[i] is EXP: LdrInitializeThunk (68)
the dependence degree are :0
ins[i] is 0x7C901166: 8D442410          LEA     EAX,[ESP+0x10]
the dependence degree are :5
ins[i] is 0x7C90116A: 89442404          MOV     DWORD PTR [ESP+0x4],EAX
the dependence degree are :64
ins[i] is 0x7C90116E: 33ED              XOR     EBP,EBP
the dependence degree are :3
ins[i] is 0x7C901170: E9029F0100        JMP     0x7C91B077      ; (*+0x19F07)
the dependence degree are :71
```

Fig 7: Instruction sequence analyzer scheme1&3 output

V. CONCLUSION

The enhanced taint flow analysis incorporates the analysis of implicit taint flow and improves the detection effectiveness of buffer overflow attack blocker .The signature free, static analysis based defense mechanism offer protection against zero-day attacks and anti-emulation obfuscation techniques.

REFERENCES

[1]W. Robertson, C. Kruegel, E. Kirda, D.Mutz, andG.Vigna. Polymorphic Worm Detection using Structural Information of Executables. In Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection RAID, 2005.

[2]Qinghua Zhang, Douglas S. Reeves, Peng Ning, S. Purushothaman Iyer.2007, "Analyzing Network Traffic to Detect Self-Decrypting Exploit Code", In Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS), pages 4-12.

[3] Ramkumar Chinchani, and Eric van den Berg.2005, "A Fast Static Analysis Approach to Detect Exploit Code inside Network Flows", Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID).

[4] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. Jan.-March 2010, "SigFree: A Signature- Free Buffer Overflow Attack Blocker", IEEE Transactions on Dependable and Secure Computing, Volume 7, Issue 1, On Page: 65 – 79.

[5] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, Peng, (2008) "STILL: Exploit Code Detection via Static Taint and Initialization Analyses", Computer Security Applications Conference (ACSAC), on page(s): 289 – 298.

[6]T. Berners-Lee, L Masinter, and M. McCahill, Uniform Resource Locators (URL), RFC 1738 (Proposed Standard), updated by RFCs 1808.