"Yukti": A Dynamic Agent based IDS with Suspect Engine to Detect Diverse XSS Attacks

K.Sivakumar
Department of E&CE
IIT, Roorkee
India

Anil.K.Sarje
Department of E&CE
IIT, Roorkee
India

K.Garg
Manipal Inst.of Technology
Manipal
India

ABSTRACT
Injecting malicious script through links, URLs (Unified resource locator) or as user inputs and getting it executed (when inputs are not validated) in the client side is called cross site scripting (XSS) attack. It is called XSS because the script that is executed here is not originated from the same client or from a trusted server. Our solution "Yukti" is devised to detect these application specific XSS attacks at network level by deep packet inspection in the live environment. Existing solutions do static security code review or scans the application for known attack patterns. "Yukti' is dynamic as the suspect engine in the solution is unique and has the capability to suspect a new attack pattern. If the suspect is analyzed to be true, the rule that would detect the attack is built into rule base dynamically. This paper discusses the design, components, architecture, dependencies, techniques, implementation and analysis of results obtained. Our results show that out of huge test cases (70000- both XSS and Non XSS) the solution is able to detect 28546 numbers of XSS attacks initially (before appending new rules in detection engine). After appending new rules based on recommendations from suspect engine, it is able to detect 32363 XSS. Yukti demonstrates considerable improvement in the performance when analyzed with leading IDS engine SNORT while detecting XSS attacks

General Terms
Web Security, Vulnerability Assessment & Intrusion Detection.

Keywords
Cross Site Scripting, XSS, Web Application Security, Application Intrusion Detection, Security Attacks and Vulnerability Management.

1. INTRODUCTION
Vulnerability is a weakness in software, operating system or hardware that can be exploited by an attacker. An exploit is a technique or software code (often in the form of scripts) that takes advantage of a vulnerability or security weakness in a piece of target software [12, 13, 14, and 16]. Vulnerabilities are of significant interest when a program containing the flaw operates in a networked environment or has access to the Internet. When vulnerabilities are discovered, disclosed and exploited, they give rise to individual and large-scale attacks challenging the security of the Web.

Cross-site scripting attacks occur when dynamically generated web pages display input that is not properly validated [1, 7, 8, and 10]. This allows an attacker to embed malicious JavaScript code into the generated page and execute the script on the machine of any user who access that site. According to [8, 13, and 15] there are three fundamental types of XSS: stored, reflected and DOM (Document object Model) based.

Stored XSS works if a HTML page includes data stored on the Web server (e.g. from a database) that originally comes from user input. Some part of a HTTP request (usually a URL parameter, cookie or the referrer location) is reflected by the web server into the HTML content that is returned to the requesting browser in reflected XSS. Reflected means that the input is written back unaltered. DOM-based XSS is very similar to the second type. A key difference is that the attack code is not embedded into the HTML content sent back by the server. Instead, it is embedded in the URL of the requested page and executed in the user's browser by faulty script code contained in the HTML content returned by the server.

Attackers often perform XSS exploitation by crafting malicious URLs and tricking users to clicking them. These links cause client side scripting languages (VBScript, JavaScript, etc.) of the attacker's choice to execute on the victim's browser. There are numerous ways to inject JavaScript (any script) code into URLs for the purpose of a XSS attack [1, 6, 8, 10, and 11].

In this paper we build our solution named "Yukti" and discuss about the state of art component Suspect Engine. Yukti provides the mechanism to dynamically detect XSS attacks at packet level using centrally grown incremental rule base.

Works related to XSS detection are discussed in section 2. Ours contribution in building the solution is discussed in section 3. In section 4, implementation of the solution and theory behind the state of the art Suspect Engine are discussed. Analyses of the results are made in section 5. Limitations & Future scope to our approach is listed in sections 6. Conclusion is drawn in section7.

2. RELATED WORK
Kirda et al [1] identify, that the code in JavaScript is vulnerable to XSS attacks and a client side solution is necessary to detect the vulnerabilities. The authors suggest a personal web fire wall NOXES that acts as a web proxy. It utilizes automatically generated rules in addition to manual ones for policing. NOXES provides an additional layer of protection which allows users to exert control over connections that browsers are making.

According to Vogt [2], dynamic Data tainting is necessary in JavaScript Engine of Mozilla Fire Fox, such that sensitive information shall not be transferred by XSS code without the user's consent. In [3] the authors recognize that the injected

malicious JavaScript through the user's web browsers (Mozilla) could create enormous damage to the site. They have proposed a solution by auditing Java Script dynamically during execution, combined with IDS (Intrusion Detection System) to detect malicious JavaScript code

In [4] the researchers have exposed the SQL injection and XSS attacks in the IE Framework. IE (Internet Explorer) is the target of most of the attacks. The authors propose a complete crawling of the site and recommend a Black Box testing using WAVES (Web application Vulnerability and Error Scanner) after doing a Reverse Engineering of the site.

In [9] the authors view the client's information as the main target for XSS attacks (such as, the cookie and the data in the hidden field). Such attacks use cookies-based session management to steal dynamic information without the user's knowledge. Client side (rather than Server side) automated IDS via central repository are the suggested solution. IDS use two servers, one for detection/collection (Proxy) and the other for Database.

According to Kruegel et al [10], it is not possible to maintain the misuse type IDS (IDS are categorized basically into misuse and anomaly detection) due to large dynamic signatures in an everyday attack scenario. The authors in [11] have identified the XSS vulnerabilities in server pages. Two basic techniques to accomplish XSS attacks in server pages include insertion of malicious code in the database and executing a link containing the malicious code itself. The approach used by the authors to detect and confirm the attack includes static analysis to detect web application vulnerabilities and dynamic analysis to check actual vulnerabilities.

After making an in depth survey on the existing solutions [1, 5, 6] we have learned that a portable solution to detect XSS attack at both server and client end is necessary. Following section discusses our contribution in building the solution.

3. OUR CONTRIBUTION

We have christened our solution as "Yukti" a Sanskrit word means trick, tactics, strategy, deduction from circumstances, combination, union, induction, junction, reasoning, plan and proof. In adherence to the name, our solution includes all the said activities. The concept, components, architecture, implementation and analysis of the solution are discussed in detail in the following section.

3.1 Solution Concept

We, in our solution, have developed a detection methodology that is based on dynamic intrusion detection coupled with agent based sensors that are deployed at both server end and client end. Yukti provides the mechanism to dynamically detect XSS using centrally grown incremental rule base.

3.1.1 How Different Is Yukti

- Yukti
  Suspect Engine is the unique component in our solution that suspects an attack and puts it into analysis. Suspect Engine helps to reduce false positives dynamically.
- Existing Other Solutions
  Such feature is not yet found in any of the existing solutions. False positives are eliminated manually after report generation only.

- Yukti
  Jpcap (external library) is used to capture the packets alone. We have written our own custom code for interpreting and extraction. It gives better control over filtering the required request, response traffic based on different parameters.
- Existing Other Solutions
  Much dependent on external libraries in turn lesser control over traffic.

- Yukti
  Detection through deep packet inspection for XSS
- Existing Other Solutions
  Detection is through static code review (HP Fortify, Appscan Source and etc.) or application vulnerability assessment using security testing (Appscan, HP Webinspect, Acunetix, Hailstorm and etc.). Some modern threat management devices do packet inspection with very limited XSS detection capability (Snort)

- Yukti
  Dynamic rules (rule building capability) and specially crafted regex using phrase structures are used to detect XSS attacks.
- Existing Other Solutions
  Static rule sets and regex are used for XSS vulnerability detection

- Yukti
  Zero day XSS based attacks can be detected with the help of Suspect Engine and Knowledge aggregator.
- Existing Other Solutions
  Zero day XSS attacks are either undetected or detected only with help of special paid services offered by vendor's 24X7 research team

- Yukti
  Attacker profiler is a value added component in our solution to keep track of the origin of attack, whereabouts of attacker.
- Existing Other Solutions
  Yet to find such a comprehensive component attached to an existing XSS detection tool.

.

3.2 Solution Components

As given in Fig 1, Yukti Intrusion Detection Server (YIDS) and Agent (YIDA) are the basic building blocks of our solution. Many other components were appended to the solution based on the necessity. Components of Yukti are listed in Fig 2. YIDS is the core component. It is comprised of YID Manager (YIDM), Agent Manager, Suspect Manager and Database Manager.

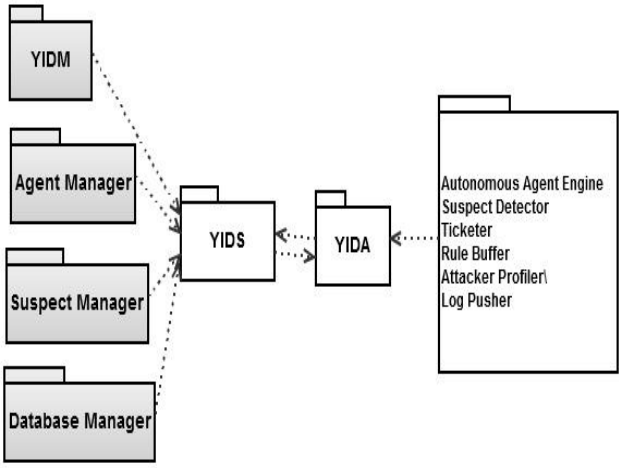

Fig 1: Building Block of Yukti

3.2.1 YIDM

YIDM includes Packet Capturing Engine (PCE), Intrusion Detection Engine (IDE), Rule Engine (RUE), Knowledge Aggregator and Dashboard. Agent Manager includes Agent Registration, Inactive Agent intimation, Agent Memory Manager and Log Manager. Suspect Manager includes Suspect Receiver, Suspect Analyzer, Rule Engine and Graphical Interface. Database Manager includes Memory Alert, Data Backup and Database cleanup components.

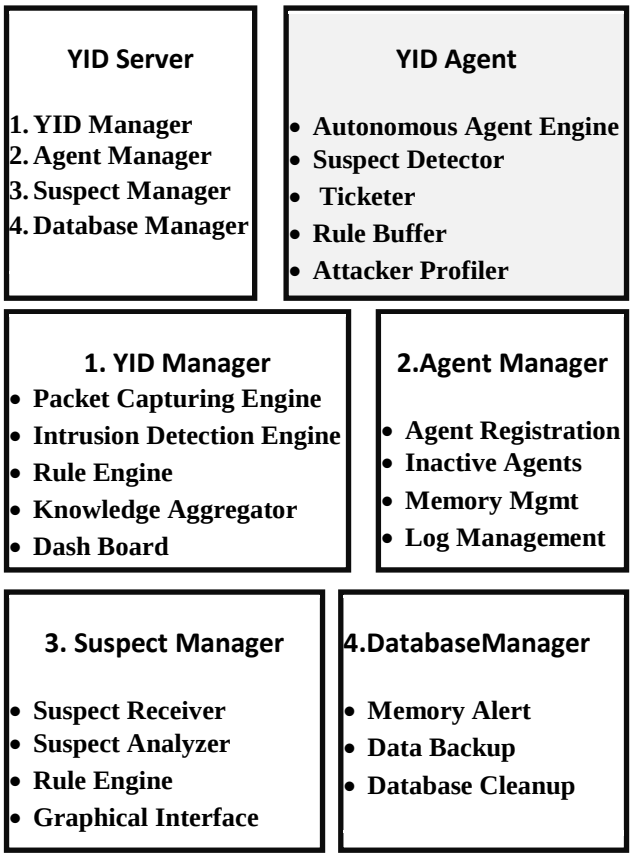

Fig 2: Components of Yukti

Rule engine is comprised of comprehensive rule database with more than 132 rules. It is a component developed out of our's continual research on XSS exploits. PCE, IDE and RUE are explained in detail in the implementation section. Knowledge aggregator is the feature through which YIDM administrator gets the latest information on recent XSS exploits and attack signatures. Aggregator is interfaced with different RSS feeders and security advisories providers. Dashboard is the feature that displays the attack statistics like the number of packets captured, categories of packets, number of attacks detected and number of suspect packets. Graphical representations of the statistics are also available for easy understanding.

3.2.2 Agent Manager

Agent Manager is a feature, we have included after feeling its necessity. Registration process is enabled once an agent is deployed in the host. This helps the YIDS to keep track of the number of hosts connected to it. Further, agents that do not communicate for a long period are verified for their active status at regular intervals. In case any agent does not respond, its inactive status is intimated to YIDS administrator through the Agent Manager for further actions.

It is necessary to keep the memory occupied by the agents as little as possible. Archived suspects, attack profiles increase the memory usage of agent. Hence the outdated archives are called back by the agent manager and stored in the server's database. Every activity carried out by the agent is recorded in the agent itself. At regular interval they are pushed back to the agent manager by the agent's log pusher. These logs are very useful for tracing back the attacks and suspects. Log management features of Agent Manager Handles this activity.

3.2.3 Suspect Manager

Suspect Manager(SM) is the unique component introduced by us in this solution. When an agent suspects a XSS intrusion, it sends the suspected information to Suspect Receiver of SM. Suspect analyzer is a mix of automated and manual analysis engine. More details of the analyzer are provided in Section 3.4 under suspect mode.

3.2.4 Database Manager

Database Manager of YIDM is used to learn the health of the database memory. As it is vital for storing archives and current information, necessary alerts are triggered when memory is nearing to 90% of the total capacity. Facilities to take regular backup and clean up are provided by DB manager.

3.2.5 YIDA

YID Agent (YIDA) is comprised of Autonomous agent engine, Suspect Detector, Ticketer, Rule Buffer, Attack Profiler and Log Pusher. Ticketer is a ticketing utility embedded within the agent to have a ticket raised in case of suspect and send it to the YIDS. More information on components and operations of YIDA are discussed in section 3.4 and in section 4.2.

3.3 Solution Architecture

YIDS is the central storage, command and control station as show in Fig 3. YIDS have an in built YIDA to protect itself from XSS attacks. For a simple version we have MySql database in the same system itself. In large environments database can be

resident on a separate server. YIDAs are the autonomous software agents that are deployed into the nodes to be protected.

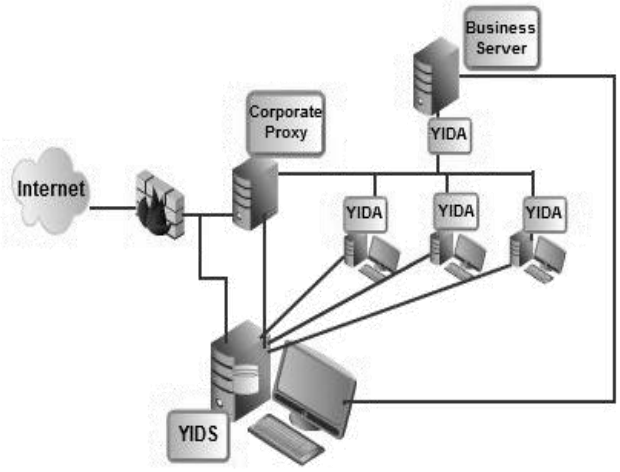

Fig 3: Architecture of YUKTI

We recommend deploying YIDA in the Business Server or Application server to detect any persistent and reflected XSS attack at first instant. Placing the YIDA at client node would detect DOM based XSS attacks and non-persistent attacks.

3.4 YIDA – YIDM Transaction Sequence

YIDA works in two modes, viz.,

1. Detect Mode
2. Suspect Mode

The YIDA registers itself to the YIDM. Upon registration YIDM pushes the updated rule sets to the YIDA. The IDE in autonomous agent engine of YIDA sniffs the packet passing through (both inward and outward), i.e. both request and response to and fro from the host device. The advantage here is YIDA does not need to hold a database for storing rule sets. These rule sets are made available as a flat file to YIDA.

Whenever detection engine is able to detect the attack, they are highlighted in the interface menu. User can know more information about the attack, rule applied by clicking the packet highlighted in red in the interface menu. In addition to the detection, the details of attacker's IP, whereabouts, whois information are collected by submitting the IP to an external whois server [19]. This information is stored by pushing into attacker's profile database in YIDS for future reference.

As given in the Fig 4 , in the suspect mode, when YIDA is unable to confirm that (packet decode) as an attack, a ticket is created. Ticket includes the copy of the packet trace and is sent to the YIDM's suspect manager. The YIDM administrator gets the alert for the ticket. The ticket is analyzed for any new type of XSS attack signature. With the help of Knowledge Aggregator and his personal experience, administrator (or an authorized user) categorizes it as an attack or not. If it is an attack signature, it is appended to the YIDM's rule data base. An updated rule set is pushed to all the YIDAs. With these current rule sets any new or variants of XSS attacks could be detected by any participant agent. In addition to knowing the concepts and components it is interesting to know about the implementation. Following section discusses the key entities related to implementation of the solution.

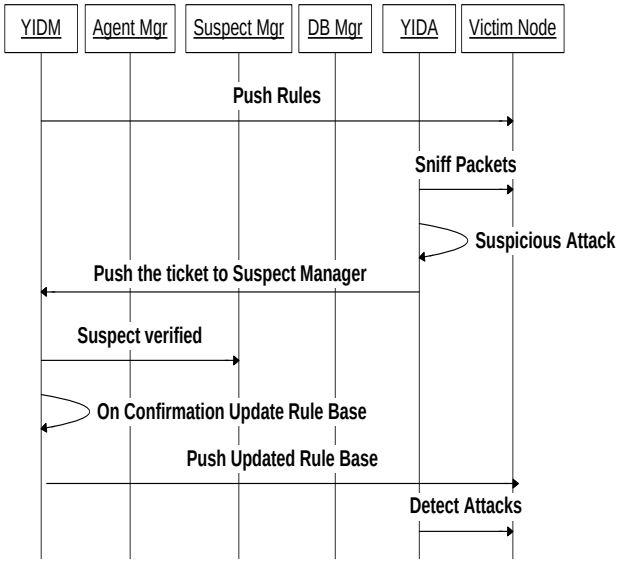

Fig 4: Sequence diagram for YIDA-YIDM in suspect mode

4. SOLUTION IMPLEMENTATION

Given below are the requirements for our development environment: The entire solution is built using Java. The portability, network adaptability, interoperability and platform independence features of Java has enabled us to develop this solution.

- Jdk 1.6 and above
- Jpcap
- Winpcap
- Mysql 5.0 and above
- Mysql Connecter - java 5.1.10 bin
- Apache Tomcat v6

4.1 Packet Inspection Flow Diagram

The flow chart given in Fig. 5 explains the packet inspection activity which is the core one in this implementation. The process is initiated as a thread (initCapturePacket). Packets are captured using the jpcap external library[20] that is called in by our application. IDSengine receives the packets and are extracted for protocol filtration by the PacketExtractor. Copies of the packets are stored in packet database. Packet contents are factored to detect whether they have any XSS signatures. This is done by comparing every rule in rule base with the decoded contents of the packet. An alert is rendered by XSSFinder if an exact match is found.

Based on the source and destination IP address of the packet, they are categorized as request or response. After classification they are stored in request and response table respectively. These packets are numbered and displayed in the table format (header and rows) in the display menu. The packets that carry the signature (those are parsed true while comparing) are highlighted in red color. By clicking the highlighted row the

administrator or user can view the decoded contents of the packet header and messages. The signature part was also highlighted for better viewing. Different decoding and encoding methods are applied depending upon the location (user input or URL) the XSS signature exists.

When XSSFinder cannot find an exact match, but detects some traces of the XSS signature, then it goes into the suspect mode and suspect engine is called. With the help of personal intelligence and knowledge obtained from the knowledge aggregator the administrator or user decides whether the suspect is a true XSS attack signature or not. If it is decided true a new regex will be created for the attack and sent to the rule base for updating and distribution to all agents (not shown in the figure).

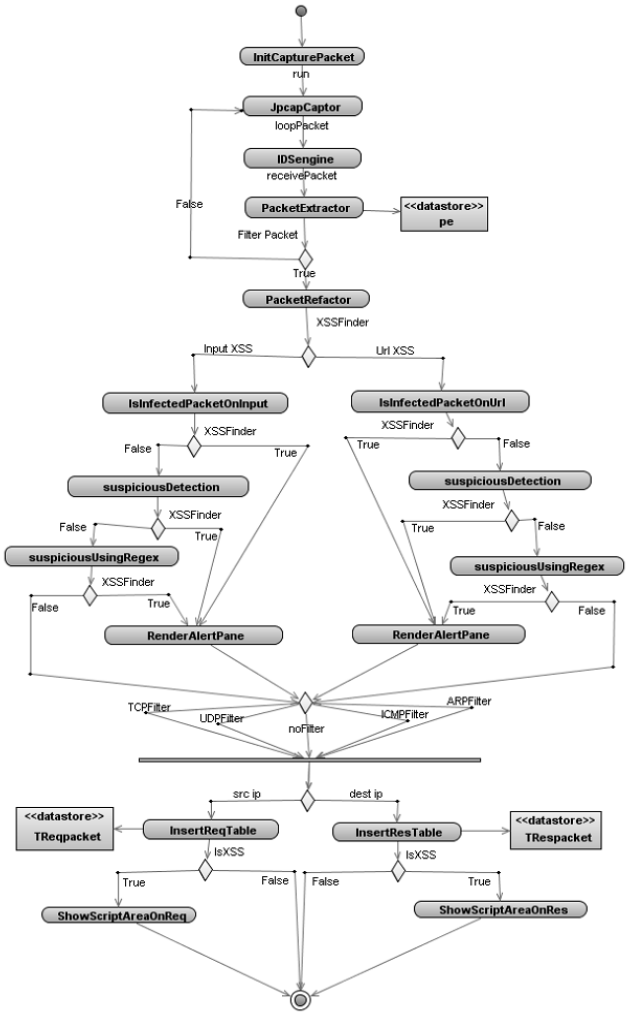

Fig 5: Packet Inspection Activity Diagram

By applying the updated rule set XSSfinder detects any such attacks if it is repeated in future. We have used rule base, rule set interchangeably in many places because we mean rule base when the rules are in database. When these rules are transferred to a flat file and used by the agent, we call it as a rule set. Important dependencies are discussed in this implementation section. The following section discusses the state of the art suspect engine of our solution.

4.2 State of the Art –Suspect Engine

The unique suspect engine in our detection system uses the following state of the art phrase-structure. It helps to achieve greater accuracy in categorizing the arriving packets into suspects or not. Table 3 lists some of the regex that are used to identify suspects. We have taken the suspect rule "SCRIPT_ALERT" to explain the grammar and its production. The regex

[<](s).?(c).?(r).?(i).?(p).?(t)[>](.*\\s)*?(alert)[(]['\"`]?.*?['\"`]?[)][;]?

is broken into smaller groups as given in the Table 1

Table 1 Breaking the regex into smaller groups

SCRIPT_ALERT	
GROUP NO	**REGEX RULE DATA**
GRP-01	[<](s)
GRP-02	.?(c)
GRP-03	.?(r)
GRP-04	.?(i)
GRP-05	.?(p)
GRP-06	.?(t)[>]
GRP-07	(.*\\s)*?
GRP-08	(alert)
GRP-09	[(]['\"`]?
GRP-10	.*?
GRP-11	['\"`]?
GRP-12	[)][;]?

The operations between each group are as given below. This is derived from our continual research.

(GRP-01 ∨ GRP-02 ∨ GRP-03 ∨ GRP-04 ∨ GRP-05) ∧ (GRP-06 ∨ GRP-07) ∧ (GRP-08) ∧ (GRP-09 ∨ GRP-10 ∨ GRP-11 ∨ GRP-12)

The grammar G for the for the expression is defined as

G={V_N ,V_T, S, P} , Where

V_N = {A_1, A_2, A_3,… A_{12}} is a finite set of non-terminal symbols of a vocabulary V, which can be replaced by other symbols..

V_T = {a_1, a_2, a_3,… a_{12}} is a finite set of terminal symbols of V, which cannot be replaced by other symbols

S is a start symbol

P is the set of productions (grammatical rules) each of the form $w_1 \rightarrow w_2$, where w_1 is a single non-terminal symbol and w_2 is a single terminal or a terminal followed by a non-terminal.

P = {S→ a_1A_1 | a_2A_2 | a_3A_3 | a_4A_4 | a_5A_5 | a_6A_6 | a_7A_7,

A_1→ a_2A_2 | a_3A_3 | a_4A_4 | a_5A_5 | a_6A_6 | a_7A_7,

A_2→ a_1A_1 | a_3A_3 | a_4A_4 | a_5A_5 | a_6A_6 | a_7A_7,

$A_3 \rightarrow a_1A_1 \mid a_2A_2 \mid a_4A_4 \mid a_5A_5 \mid a_6A_6 \mid a_7A_7,$

$A_4 \rightarrow a_1A_1 \mid a_2A_2 \mid a_4A_4 \mid a_5A_5 \mid a_6A_6 \mid a_7A_7,$

$A_5 \rightarrow a_1A_1 \mid a_2A_2 \mid a_3A_3 \mid a_4A_4 \mid a_6A_6 \mid a_7A_7,$

$A_6 \rightarrow a_7A_7 \mid a_8A_8,$

$A_7 \rightarrow a_6A_6 \mid a_8A_8,$

$A_8 \rightarrow a_9A_9 \mid a_{10}A_{10} \mid a_{11}A_{11} \mid a_{12}A_{12} \mid a_9 \mid a_{10} \mid a_{11} \mid a_{12},$

$A_9 \rightarrow a_{10}A_{10} \mid a_{11}A_{11} \mid a_{12}A_{12} \mid a_{10} \mid a_{11} \mid a_{12},$

$A_{10} \rightarrow a_9A_9 \mid a_{11}A_{11} \mid a_{12}A_{12} \mid a_9 \mid a_{11} \mid a_{12},$

$A_{11} \rightarrow a_9A_9 \mid a_{10}A_{10} \mid a_{12}A_{12} \mid a_9 \mid a_{10} \mid a_{12},$

$A_{12} \rightarrow a_9A_9 \mid a_{10}A_{10} \mid a_{11}A_{11} \mid a_9 \mid a_{10} \mid a_{11}\}$

Here in this example $\{A_1 \ldots A_{12}\}$ are our {GRP-01 … GRP-12}, $\{a_1, \ldots a_{12}\}$ are {<,s,c,r…..),;} and S →<. A high level explanation is provided above, where as each groups follow their regex syntax to parse further at next level.

5. RESULT ANALYSIS

We have taken a large set of test cases (attack scripts) to conduct an extensive analysis of our solution. Well known XSS exploits from different sources [8, 12, 16, 17 and 18] and huge collection of cases derived from the archives of xssed.com (http://www.xssed.com/archive) were used for analysis. A web application was created to access all these test cases. Test cases were hosted in web environment. They were accessed as Form inputs, email links and etc. The process of passing these test cases as an input to the web application has been automated.

Results are tabulated in Table 2. First column "TestCases" gives the number of test cases that are given as input. B–means the rule set before applying any new rules. A-means the rule set after applying new rules. "NOD B" represents the number of attacks detected using rule set B.

"RULES B" gives the number of rules that exactly matched with the XSS signature in the input. "SUSP B" gives the number of signatures that are suspected as attack while using rule base B. "NOUD B" represents the number of undetected cases. "NOD A" represents the number of attacks detected using rule set A. "RULES A" gives the number of rules that exactly matched with the XSS signature in the input. "SUSP A" gives the number of signatures that are suspected as attack while using rule base A. "NOUD A" represents the number of undetected cases.

There are 70101 test cases (appx. 15% are invalid cases) used and 28546 were detected using 132 rules in rule set B. 2525 cases were suspected. One important point to be noticed here is that it is not that every time a new rule is used for matching purpose. Same rule can be used for detecting several cases. Accordingly there are only 132 rules available in the rule base B. Rule 1 is used for detecting 1 case, rule 2 is used to detect 346 cases, rule 3 is used to detect 23680 cases.

The summary of the analysis is, around 28546 attacks were detected before applying any new rules whereas 32363 detection were made after updating the rule base with new rules. Similarly numbers of suspects were increased from 2525 to 6251 after applying new rules. This is due to the fact that new rules would help in creating new regular expressions that are helpful to suspect additionally. Here also it is not that 2525 suspect rules are used to detect. One suspect rule (regex) can be used to suspect several cases. Accordingly it was observed that there are only 5 suspect rules . Rule 2 is used to detect 14 cases before applying any new rule. Rule 4 and 5 were used to suspect 1306 and 1205 cases respectively. It could also be observed that slighter modification or appending new suspect rule has helped in suspecting more cases. Samples of suspect rules are given in Table 3. It has three columns. First column lists suspect rule number, second column lists the the suspect rule name and the third one gives the exact syntax that comprised of regular expressions. Initially there were only 2 suspect rules, but it is appended with more suspect rules to help detection

Table 2 Suspected and Detected XSS Attacks

TestCases	RULES B	SUSP B	NOD B	NOUD B	RULES A	SUSP A	NOD A	NOUD A
101	54	15	69	32	61	22	83	18
1-1000	467	27	494	95	468	69	537	52
1001-2000	356	1	357	253	356	151	507	103
2001-3000	749	38	787	13	749	49	798	2
3001-5000	1110	81	1191	181	1110	220	1330	42
5001-6000	542	23	565	97	543	48	591	71
6001-8000	904	127	1031	288	922	210	1132	187
8001-10000	723	118	841	187	725	172	897	131
10001-12000	692	70	762	149	692	115	807	104
12001-14000	858	53	911	254	858	128	986	179
14001-16000	800	97	897	384	800	238	1038	243
16001-18000	936	315	1251	172	937	360	1297	126
18001-20000	884	198	1082	199	892	223	1115	166
20001-22000	698	83	781	312	699	93	792	301
22001-24000	988	58	1046	159	990	106	1096	109
24001-26000	920	58	978	243	921	163	1084	137
26001-28000	932	34	966	180	932	73	1005	141
28001-30000	689	27	716	428	691	178	869	275
30001-32000	678	57	735	445	678	237	915	265
32001-34000	672	189	861	412	673	407	1080	193
34001-36000	686	114	800	367	686	198	884	283
36001-38000	622	46	668	545	626	255	881	332
38001-40000	917	52	969	351	918	230	1148	172
40001-43000	1209	118	1327	534	1214	268	1482	379
43001-46000	1265	85	1350	548	1273	394	1667	231
46001-49000	1247	50	1297	560	1250	240	1490	367
49001-52000	1118	76	1194	711	1132	456	1588	317
52001-55000	1275	34	1309	395	1276	188	1464	240
55001-58000	897	76	973	336	900	215	1115	194
58001-61000	653	78	731	244	653	174	827	148
61001-64000	636	72	708	255	637	159	796	167
64001-67000	633	42	675	346	639	162	801	220
67001-70000	211	13	224	142	211	50	261	105
TOTAL	26021	2525	28546	9817	26112	6251	32363	6000

Snort IDS [21] has more than twenty thousand signatures to detect all types of attack including network and other application layer attacks. In contrast it has lesser number of rules (< 100 non repetitive) to detect XSS attacks. When the interpreted Snort rules for XSS were used to detect the test cases explained above, we found considerable numbers of false positives and false negatives. The ever changing camouflages of XSS attacks are reason for increased true negatives. Generalized content matching for the text like "script" is causing more false positives. Snort is again categorizing some of them as "cross site attempt". Commercial version of snort that provides subscribed rule sets could include these new updated signatures, thus creating the dependency in pushing the new rules from external source. Yukti's capability to suspect XSS attack using complex regex (unlike simple pattern matching in Snort for XSS) makes it unique in building the dynamic rule set and suspect rules

instantaneously within the system itself. It is evident from the results displayed in the table that our solution is able detect XSS attacks effectively and can improve its performance dynamically on its own. Samples of XSS attack signatures are given in the Table 4. It has four columns. First gives the rule number, second rule name, third column gives the rule pattern for the XSS attack when it is given as the user input. Fourth column gives the rule pattern to detect the XSS attack when it is exploited via URI. Fig. 6 shows the packet 24 that is suspected for XSS attack.

Table 3. Suspect Rules

SUSP NO.	SUSPECT RULE NAME	SUSPECT RULE (Regex)
1	EVENT_VB_MSGBOX	(on.*?)[=](vbscript)[:](msgbox)[(][\"`]?.*?[\"`]?[)]
2	EXP_EVAL_ALERT	(expression)[(](eval)?[(]?[\"`]?(alert)[(][\"`]?.*?[\"`]?[)][\"`]?[)]?[)]
3	ALERT_STR_CHARCODE	(alert)[(][\"`]?(string)[.](fromcharcode)[(][0-9]{1,2}[,][0-9]{1,2}[,][0-9]{1,2}[)][\"`]?[)]
4	SRC_JS	(src)[=][\"`]?((https?|ftp|gopher|telnet|file|notes|ms-help):((//)|(\\\\))+[\\w\\d:#@%/;$()~_?\\+-=\\\\.&]*)[.](j)[^pg][s]?[\"`]?
5	JAVASCRIPT_EVAL_ALERT	(data)?(on.*?)?(background)?(src)?(href)?[=](3d)?[\"`]?.?(j)?.?(a)?.?(v)?.?(a)?.?(script)?[:#]?(eval)?[(]?[\"`]?(alert)[(][\"`]?.*?[\"`]?[)][\"`]?[)]?[;]?[\"`]?
12	JAVASCRIPT_ALERT	(data)?(on.*?)?(background)?(src)?(href)?[=](3d)?[\"`]?\\s.\\s.?(j)?.?(a)?.?(v)?.?(ascript)?[:#]?(alert)[(][\"`]?.*?[\"`]?[)]
13	**SCRIPT_ALERT**	**[<](s).?(c).?(r).?(i).?(p).?(t)[>](.*\\s)*?(alert)[(][\"`]?.*?[\"`]?[)][;]?**

Table 4. Samples of XSS Attack Signatures

RULE NO.	RULE NAME	INPUT RULE PATTERN	URL RULE PATTERN
2	SCRIPT src attack	<SCRIPT SRC=http:::.js>	%3CSCRIPT%20SRC%3Dhttp:::.js%3E
3	SCRIPT alert attack 1	<SCRIPT>alert(:::)</SCRIPT>	%3CSCRIPT%3Ealert(:::)%3C/SCRIPT%3E
4	SCRIPT alert attack 2	%3Cscript%3Ealert(:::)%3C%2Fscript%3E	%3Cscript%3Ealert%28:::%29%3C%2Fscript%3E
7	XSS Attack 3	%3CSCRIPT%3Ealert%28%2F:::%2F.source%29%3C%2FSCRIPT%3E	<SCRIPT>alert(/:::/.source)</SCRIPT>
8	XSS Attack 4	string%20fromcharcode%2888%2C83%2C83	string%20fromcharcode(88,83,83
11	IMG XSS using JavaScript 3	%3CIMG+SRC%3D%22jav%09ascript%3Aalert%28:::%29%3B%22%3E	<IMG SRC=\"jav ascript:alert(:::);\">
40	BODY XSS Attack 1	%3CBODY+BACKGROUND%3D%22javascript%3Aalert%28:::%29%22%3E	<BODY BACKGROUND=\"javascript:alert(:::)\">

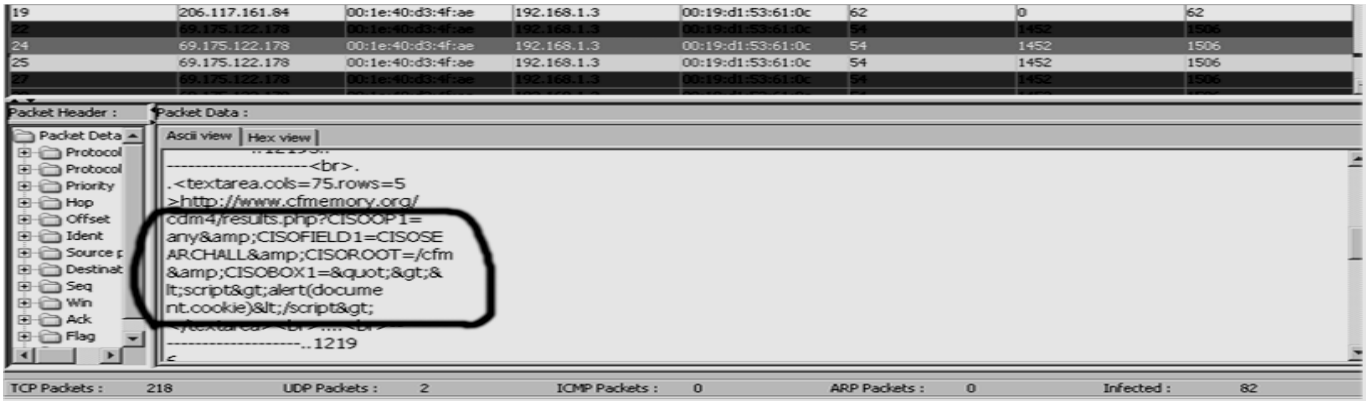

Fig 6: Packet 24 - Identified as Suspicious

6. LIMITATATIONS & FUTURE SCOPE

As much as possible we have populated our rule base with all updated rules to detect latest variants of XSS attack. Still we find attackers are trying with new circumvents. Our solution is able to capture them as suspects but not as an attack at first instant. But by analyzing the suspect and updating the rule base such attack can be detected from next attempt. Reasonable attempt was made to detect the attack even when attackers try to do application evasion techniques. But our solution has limitation in detecting network evade (if exists) XSS attacks.

Though the detection is done only to XSS based attacks, our model is portable and compatible to accommodate any other network protocol and payload based attacks. In future much scope is there to extend the solution to include all such attack detection. Further "Yukti" is currently limited to XSS detection in this phase I, whereas in phase II, preventing XSS attacks using proxy based agents is included in the scope.

7. CONCLUSION

Day by day more and more websites are identified for XSS exploitation. It is a challenge for enterprises and individuals to keep them safe from new circumvents of XSS attacks. The damage could range from stealing confidential information of client and to the extent of penetrating into the corporate network. A scalable solution that is independent of type of browser, platform and architecture is need of the time. "Yukti" is designed to be portable and scalable. The architecture, component requirement, dependencies and implementation that are discussed in this paper will enable to build any new attack detection solution for any variants of network and scripting based attacks.

8. REFERENCES

[1] E.Kirda, C.Kruegel, G.Vigna, and N.Jovanovic,"Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks"SAC'06 April 23-27,2006, Dijon, France.

[2] P.Vogt, "Cross Site Scripting (XSS) attack prevention with dynamic data tainting", 2006

[3] O. Hallaraker and G.Vigna," Detecting Malicious JavaScript Code in Mozilla ", Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)

[4] Y.Huang, C.Tsai, T.Lin, S. Huang, D.T. Kuo', "A testing framework for Web application, security assessment", Computer Networks 48 (2005) 739–761, ELSEVIER

[5] M.Egele, M.Szydlowski, E. Kirda, and C. Kruegel,"Using Static Program Analysis to Aid Intrusion Detection", Austrian Science Foundation (FWF) under grant P18368-N04

[6] N.Jovanovic, C. Kruegel and E.Kirda," Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities", Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)

[7] F.Valeur, G.Vigna, C.Kruegel, E.Kirda, "An Anomaly driven Reverse Proxy for Web Applications", SAC'06 April 2327,2006, Dijon, France, ACM 1595931082/ 06/0004

[8] K. Sivakumar & K. Garg "Constructing a "Common Cross Site Scripting Vulnerabilities Enumeration (CXE)" Using CWE and CVE", Lecture Notes in Computer Science, Springer Berlin / Heidelberg, Volume 4812/2007, 277-291

[9] O.Ismaill, M.E.Youki, K.adobayashi, S. Yamaguch, "A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability" Proceedings of the 18th International Conference on Advanced Information Networking and Application (AINA'04)

[10] Christopher Kruegel, G. Vigna, William Robertson, "A multi-model approach to the detection of web-based attacks", Computer Networks 48 (2005) 717–738-ELSEVIER.

[11] G.A.Lucca, A.R.Fasolino et all, "Identifying Cross Site Scripting Vulnerabilities in Web Applications", Proceedings of the Sixth IEEE International Workshop on Web Site Evolution (WSE'04)

[12] "The Common Vulnerabilities and Exposures Initiative," http://cve.mitre.org/cve/

[13] "OWASP top ten Security vulnerabilities", https://www.owasp.org/index.php/Category:OWASP_Top_ Ten_Project

[14] Department of Homeland Security National Cyber Security Division's "Build Security In" (BSI) web site, http://buildsecurityin.us-cert.gov

[15] National Vulnerability database http://nvd.nist.gov/

[16] Real World XSS, http://sandsprite.com/Sleuth/papers/ RealWorld_XSS_1.html

[17] XSS cheat sheet, http:// ha.ckers.org/xss.html