Online Detect Polymorphic Exploit Based on Data Mining

Wei Wang[1] Huazhang Wang[1] Daisheng Luo[1] Yong Fang[2]

[1] Institute of Image & Information, Sichuan University, Chengdu 610065, P.R. China
[2] Institute of Security & Information. Sichuan University, Chengdu 610065, P.R. China

Abstract

In recent years, Internet worms increasingly threaten the Internet hosts and service and polymorphic worms can evade signature-based intrusion detection systems. We propose DMPolD (Data Ming Polymorphism Detection) to detect polymorphic exploit based on semantic signature and data-mining. We analyze the feature of polymorphic exploit and the feature of perfect ones. We propose a method to online detect worm through recognize JUMP address based on data-mining i.e., Bayes. To prove this idea, we implement a plug-in of Snort – ODMSnort and do the experiment on it. The evaluation results show that DMPolD can detect polymorphic exploit and has very low false-positive.

Keywords: Data-mining, Polymorphic Worms, Security

1. Introduction

In recent years, Internet worms increasingly threaten the Internet hosts and service. Toward defending against Internet worms, the research community has proposed and built intrusion detection systems (IDSes). A network administrator deploys the IDS at the gateway between the network edges. The IDS searches inbound traffic for known patterns, or signatures. The signature is a tuple (IP-proto, dst-port, byteseq), where byteseq is a variable-length, fixed sequence of bytes. When traffic corresponds to signature, the IDS may raise an alarm. To date, IDSes use fixed byteseq of signature from worm's payload. Matching techniques include string matching at arbitrary payload offsets; string matching at fixed payload offsets; and matching of regular expressions within a flow's payload. [1][2]

These systems all use fixed, contiguous substring as worm's signature and all make the same underlying assumptions that there exists a single payload substring that will remain invariant across worm traffic, and will be sufficiently to identify the worm.

But those assumptions are naïve. When the well-known attacks or worms are modified / transformed differently, the IDS might fail due to its inability to math them in signature database. We call these transformed worms as polymorphic worms. A polymorphic worm author may craft a worm that substantially changed its payload every time, and thus evades matching signature of IDS.

To detected polymorphic worms, we propose a new method to detect polymorphic worms. In this paper, our main contributions are as follows:

(1) Propose an exploit model – the OSJUMP model.

(2) Based on the OSJUMP model, we analysis the features of polymorphic exploits and features of perfect ones.

(3) We propose methods to detect exploit through recognize JUMP address based on data-mining such as Bayes. The evaluation results show that DMPolD can detect polymorphic exploits and has very low false-positive.

The rest of this paper is organized as follows. Section 2 describes the relative work. Section 3 proposes the worm attack model -- the OSJUMP model. Section 4 discusses the polymorphic exploits and perfect ones. Section 5 describes our methods. Section 6 presents experiment results and Section 8 presents discussion.

2. Related work

Honeycomb[3], Autograph[4], and Earlybird[5] are pioneers of systems which automatically generate worm signatures. They automatic capture Internet traffics, classify worm flows, and generate worm signatures, but they didn't consider the polymorphic worms.

After recognize the risk of polymorphic worms, several solutions have been proposed to detect polymorphic attacks or worms.

One approach to detect buffer overflow attack code is concentrate on the sledge (e.g. string of NOPs) of the attack. The method has been presented in [6].

NGSEC proposed several solutions to detect polymorphic worms using IDS [7]. Their solutions include Shellcode payload decryption, signatures to

detect the decrypted engine, decrypted engine emulation, and NOP section detection. They concluded that NOP section detection was the best technique.

But we found that the NOP section detection not suit for exactly attacks because NOP section can be uniform random under such situation.

TaintCheck[8] can semantic analysis overflow-attack and can be used to generate exactly signatures, but it needs to monitor CPU execution and data flow in the memory. The performance overhead is too high to suit for network environment.

Buttercup [9] is a system which uses range of JUMP address to detect overflow attack, but it's so naïve that attacker can easily select JUMP address out of the range and evade the Buttercup's check.

Polygraph[10] uses multiple disjoint content substrings as feature to detect polymorphic worms, but the content substring features are also fixed.

Motivate by those work, we propose a new method to detection polymorphic exploit by data-mining. After setting up exploit generation attacking model, we semantically analyze polymorphic exploits and use critical part as signature. The critical part is detected by data-mining such as Bayes rather than directly match.

3. Exploit model

In mostly case, the propagation of the worm is base on overflow holes and attacks. By deeply analyzing overflow holes and worm exploit code, we propose the model of exploit – the OSJUMP model, which can be used for explaining the character of worm attack.

3.1. Definition of the OSJUMP model

The OSJUMP means Overflow, Shellcode and JUMP, all of three are critical elements in a successful overflow attack.

The OSJUMP model is show as Figure 1.

The purpose of the three parts and the relationship among them describe below.

1) Overflow: Construct Shellcode and overwrite sensitive data. Overflow is the precondition of overflow attack.

2) Shellcode: Target binary code for execute. Shellcode is constructed by overflow. Successful executing Shellcode is the purpose of the overflow attack.

3) JUMP: Jump into Shellcode via system execution flow such as function return, function called or memory reallocates or free. JUMP is the most critical element for successful overflow attack.

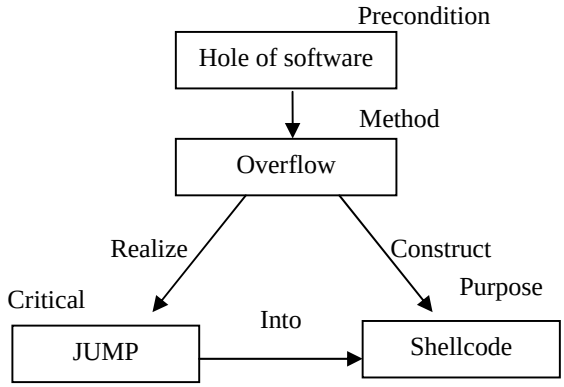

Fig. 1: The overview of OSJUMP model.

3.2. Discussion of the OSJUMP model

Based on OSJUMP model, we can model many types of overflow attacks, such as stack overflow, heap overflow, virtual function overflow, and format overflow. The results of modeling have put in Table 1.

Overflow	**Shellcode**	**JUMP**
Stack Overflow	Constructed in stack	Overwrite return IP address, JUMPed when function return
Heap Overflow	Constructed in heap	Overflow pointer of heap manage struck, JUMPed when allocate or free again
Object Overflow	Constructed in virtual function table	JUMPed when system call virtual functions
Format Overflow	Constructed in stack	JUMPed when system call functions

Table 1: Use OSJUMP to modeling overflow attacks.

The OSJUMP model can be used in many situations: overflow attacks, worm attacks, and etc. It not just suit for Windows system but also suit for *nux system. The worms have show that all of them follow the OSJUMP model. Moreover, future overflow attack technique and worm attack technique can be predicted and analyzed by the OSJUMP model too.

4. Analysis of polymorphic exploit

Thanks to the OSJUMP model, we can understand the exploit and can semantically analyze polymorphic ones.

4.1. Principle of polymorphic exploit

'Polymorphic' means something which do not has fix build. Polymorphic exploits are ones which can self-change payload every time. Polymorphic exploits exist because they can change two parts.

The 'overflow' part: this part just takes the space of memory, so the 'overflow' part can be constructed freely.

The 'Shellcode' part: this part depends on object of attacker or worm, so it can be changed or encrypted every time too.

Polymorphic exploit are generated based on those two principles. Overflow part and Shellcode part are different each time, so traditional IDS signature can't match the polymorphic exploit. An example format of polymorphic exploit shows as Figure 2.

Fig. 2: An example format of polymorphic exploit

4.2. Polymorphic engine

There have some polymorphic engines that can generate polymorphic worms automatically already. Clet[11] and ADMmutate[12] are both polymorphic engines. They can output encrypted code which is completely different each time, and a decryption routine that is obfuscated differently each time too.

But the outputs of engines have some common substrings that present in all outputs. So they can't generate perfectly polymorphic worms. Polymorphic engines can be improved and the detail analysis can be found in [10].

4.3. Perfectly polymorphic exploit

The perfectly polymorphic exploit don't have any common substrings and can be generated by following rules.

1) Overflow: this part doesn't influence the attack, so it can be filled by uniformly at random.

2) Shellcode: this part can be encrypted by different key, and decryption routine is differently each time.

3) JUMP: can be selected different value each time.

The perfectly polymorphic exploit which generated according to above rules don't have any common substring with each others. So they can evade any pattern match IDSes. Motivated by this, we propose new polymorphic exploit's detection methods -- detect by data mining.

5. Detect Polymorphic by data-mining

We describe our methods in this section and present experiment results on the next section.

5.1. Definition

Here, we want to detect exploit online.

Definition 1: F is a network flow, we want to online judge whether it is polymorphic exploit or not. So we need to create a function $R(*)$, where

$$R(F) = 0\text{; if } F = Normal$$

$$R(F) = 1\text{; if } F = PolAttack$$

Since overflow can be uniform random and Shellcode can be total different each time, we can't use they two as signature to judge polymorphic exploits. However, position of JUMP must be fixed and value of JUMP address can't be selected arbitrariness. The value only has some options in the system; otherwise the overflow attack can't successful and can't JUMP into Shellcode. For example, JUMP position of Red Code worm is at 254 byte offset.

So we propose a new method for matching exploits-- classify 4bytes substring at JUMP position of flow. If the substring is classified to be a JUMP address, we classify the flow to be a worm in succession.

Definition 2: To classify 4bytes substring at JUMP position of flow, we need create a function L(*), where

$$L(x) = 0\text{; if } x \neq JUMP \Rightarrow F = Normal$$

$$L(x) = 1\text{; if } x = JUMP \Rightarrow F = PolAttack$$

To create function L(*) is a very challenge work. We try to use data-mining such as Bayes match to complete it. The worms and JUMP address they used are show as Table 2.

WORM	JUMP
RedCode	0x7801cbd3
Blaster	0x0018759F
Nachi	0x0100139D
Sasser	0x01004600
Witty	0x5e077663
Slammer	0x42b0c9dc

Table 2: JUMP addresses used by worms.

Besides JUMP address in Talbe1, we also include some universe jump address such as 0x7FFA1571, 0x7FFA4512 in the list because they used widely in the overflow exploit.

5.2. Bayes match

Directly match JUMP address can be easily violate by select some new option ones. So we consider classify 4bytes substring by Bayes match.

Each byte of 4bytes substring is associated with a score and an overall threshold. In contrast with the exact matching, Bayes match provide probabilistic matching-- given a 4bytes substring, we compute the probability that the string is a JUMP address. If the resulting probability is over the threshold, we classify the string to be a JUMP address and the flow to be a worm.

A 4bytes substring x can be presented as

$$\{xi\}, 1 \le i \le 4$$

Let L(x) denote the true label of x, then

$L(x) = JUMP$ denotes x is a JUMP address

$L(x) =\sim JUMP$ denotes x is not a JUMP address

Thus, to classify a sample $\{x1, x2, x3, x4\}$, we wish to compute $\Pr(L(x) = JUMP \mid x)$ and $\Pr(L(x) =\sim JUMP \mid x)$

To calculate $\Pr(L(x) = JUMP \mid x)$, we use Bayes law:

$$\Pr(L(x) = JUMP \mid x)$$
$$= \frac{\Pr(x \mid L(x) = JUMP)}{\Pr(x)} \Pr(L(x) = JUMP)$$

From the independence assumption of the naïve Bayes model, we can compute this as follow:

$$= \frac{\Pr(L(x) = JUMP)}{\Pr(x)} \prod_{1 \le i \le 4} \Pr(xi \mid L(xi) = JUMP)$$

We only need to calculate the radio of $\Pr(L(x) = JUMP \mid x)$ and $\Pr(L(x) =\sim JUMP \mid x)$.

$$\frac{L(x) = JUMP}{L(x) =\sim JUMP}$$
$$= \frac{\frac{\Pr(L(x) = JUMP)}{\Pr(x)} \prod_{1 \le i \le 4} \Pr(xi \mid L(xi) = JUMP)}{\frac{\Pr(L(x) =\sim JUMP)}{\Pr(x)} \prod_{1 \le i \le 4} \Pr(xi \mid L(xi) =\sim JUMP)}$$
$$= \frac{\Pr(L(x) = JUMP) \prod_{1 \le i \le 4} \Pr(xi \mid L(xi) = JUMP)}{\Pr(L(x) =\sim JUMP) \prod_{1 \le i \le 4} \Pr(xi \mid L(xi) =\sim JUMP)}$$

We just set the value of $\Pr(xi \mid L(xi) = JUMP)$ is 0.5. Then we calculate each byte's probability and compare to the threshold.

6. Evaluation

In this section, we present out test environment and results.

6.1. Environment

In order to test our OMPOLD, we implement ODMSnort as a plug-in in Snort 2.4.2. The Program Language is C and file length is about 600 lines. We add a key word "jumpdm" in the Snort system, so the detection rule is "jumpdm: offset", where "offset" indicates offset of JUMP in the network flow. For example, the detection rule of RPC vulnerability is

alert tcp $EXTERNAL_NET any -> $HOME_NET 135 (msg:"found RPC jumpdm!"; jumpdm:916; reference:bugtraq,8205; reference:cve,2003-0352; classtype:attempted-admin; sid:1000831; rev:1;)

The detection machine which install Snort is Intel Celeron 2.26G, 256M memory, and 80G hard disk.

The attack machine is Intel P4 1.4G, 256M memory, 40G hard disk, the system type is Windows XP SP2.

The target machine base on VMWare, the system type is Windows 2000 SP1.

6.2. Detection rate

Training data: We use JUMP addresses show in Table 2 plus some universe JUMP address as training data.

Test data: We use Metaexploit attack framework, select 5 actually attack exploits for Windows environment. For each actual exploit, we produce 4 polymorphic exploits by automatically engine, and generate 5 perfectly polymorphic ones by manual. So the number of test data is 5*1+5*4+5*5 = 50 totally.

The detection rate results are showed in Table 3.

	Snort	DMPolD
Origin (5)	100%	100%
Engine (20)	85.0%	100%
Manuel (25)	45.0%	75.0%
Totally	62%	80%

Table 3: The results of detection rate.

From the Table 3, we can see that our methods can improve detection rate greatly.

For the origin exploits, each algorithm can detect correctly.

But for the polymorphic exploits which produced by engine, snort can't match it in signature database and produce 15% false negative, while our methods still produce 0% false negative.

For perfectly polymorphic exploits, snort only can detect part of them, while DMPolD can still detect most of them. DMPolD can improve 18% detection rate totally.

6.3. False positive alarm rate

As a new technique of NIDS, one key is false positive alarm should not too high. Too many false positive alarms will lead administrator to boring and can not find really attack. So, we did a lot of experiment on false positive alarm: relative false positive alarm rate, absolutely false positive alarm rate, and worst positive alarm.

Relative false positive alarm rate

Relative false positive alarm rate is defined as radio of alarm number from DMPolD to the alarm number from Snort based on the same test data. Assume result of Snort is correct, then the number of alarm will not increase after adding DMPolD plunge.

Definition 3:

$$\text{Relative false positive alarm rate} = \frac{\text{DMPolD alarm number} - \text{Snort alarm number}}{\text{Snort alarm number}}$$

We do the experiment on the MIT2000 data set. Test the origin Snort and DMPolD plunge on the same test sets. The results show as Figure 3.

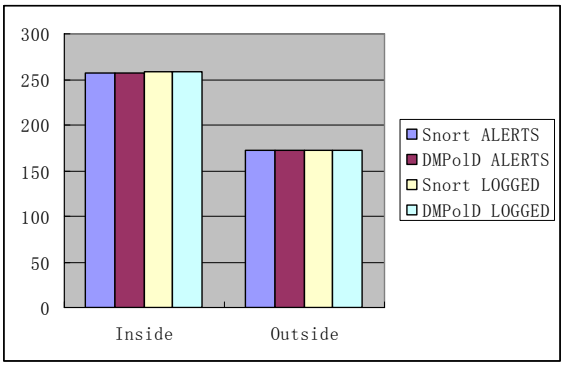

Fig 3: The result of relative false positive alarm rate

Figure 3 shows that the alarm number of origin Snort and one of DMPolD plunge are same. So, the relative false positive alarm rate is zero. It means that DMPolD plunge will not introduce relative false positive alarms.

Absolutely false positive alarm rate

Absolutely false positive alarm rate is defined as the alarm number of DMPolD on the normal flow. Because it should not introduce alarm on the normal flow, the alarm is false positive alarm.

Definition 4:

$$\text{Absolutely false positive alarm rate} = \frac{\text{Number of DMPolD alarm}}{\text{Number of normal packet}}$$

We capture the normal flow on the gateway of Institute of Image and Information, Sichuan University. We get 10220 TCP packets, 2008 UDP packets, and 69 other packets, 12297 totally. We disable all plunge of Snort except DMPolD and do the experiment on the data set. The results show as Table 4.

No. of Packet	12297
ALERT	0
LOGGED	0
Absolutely Error	0.0%

Table 4: Absolutely false positive alarm rate on normal flow.

We can find that there is no false positive alarm on the normal flow, which means DMPolD has very good efficiency.

Worst positive alarm rate

In order to investigate the result of DMPolD on all stings of normal flow, we change the implementation of DMPolD to test all the string in the flow rather than JUMP position. We call this test as worst positive alarm rate. The goal of experiment is to do the most strict false positive alarm test.

Definition 5:

$$\text{Worst positive alarm rate} = \frac{\text{DMPolD alarm number}}{\text{Number of normal string check}}$$

There are 5233170 bytes in the normal flow we get, and then test number is 5233170 * 4 = 20932680. In our test, DMPolD only raise 44 alarms, the results show as Table 5.

No. of Test t	20932680
ALERT	44
LOGGED	44
The Worst Error	0.00021%

Table 5: Worst false positive alarm.

There are only 0.00021 % positive alarm rate, which is confirm the result from [9].

6.4. Performance evaluation

We also measured the performance of detection algorithm. We record the detection time of Snort with and without OMPolD plunge. The results show as Figure 4. The results are average value of 10 experiments.

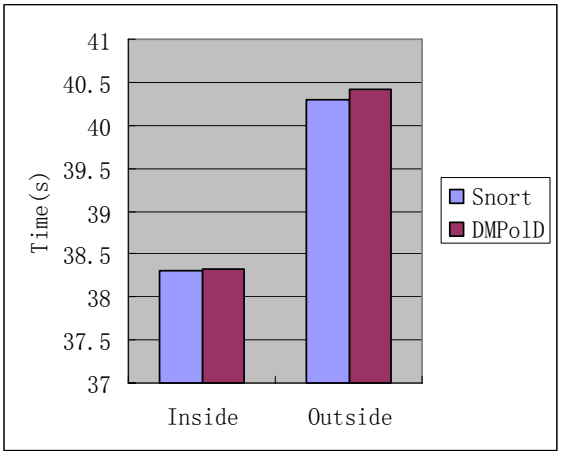

Fig 4: Average detection time

From Figure 4, we can find that OMPolD plunge only add a little performance overhead. The reason is that the new methods only need to compute the probability result of 4bytes substring at fix position and not need to do string pattern match in a large range.

7. Discussion

We can see from the above sections that DMPolD has many advantages. Here we discuss some details in DMPolD implementation.

7.1. Offset of JUMP position

Because the difference of system version (XP, 2000, hot patch), some exploits' jump positions are not the same on different systems. So, one vulnerable hole may needs multiple rules.

But in our experiments, JUMP offset is unique on the same system, and the difference is not large among different versions. So we can either create rule for each system version, or check a range of position. We can see from 6.3.3 that even worst false positive rate is very low, which means check a range of position for JUMP address also can get good results.

7.2. Rule creation

By DMPol, it's very easy to create new rule for new exploit.

Creators only need to known JUMP position in the network flow and then can write the rules. In contrast with traditional method, DMPol don't not need to complexly analyses and can reduce time cost greatly, which is very important in worm defense battle. With some Taint technical[8], DMPol can be used to generate signature automatically.

7.3. Decode

In some network applications, network flow is encoded by e.g. Base64, Unicode. And in some cases, network flow is truncated or retransferred. But there are so many plug-ins in snort can finish decode and re-construct work. So we can enjoy the effect of our OMPolD.

8. Conclusions

In this paper, we propose new methods based on data-mining to detect polymorphic exploit. Experiment results show that our methods are better than former work and can improve false negative rate and introduce nearly 0 false positive rate.

The future work will be focused on more deeply semantic analysis of worms and get more information for data-mining to improve the false negative and false positive.

Acknowledgement

The authors would like to thank great help from many anonymous.

References

[1] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31:23-24, December 1999.

[2] T. S. Project. Snort, *the open-source network intrusion detection system.* http://www.snort.org/.

[3] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. *2nd Workshop on Hot Topics in Networks (HotNets-II)*,2003.

[4] H.-A. Kim and B. Karp. Autograph: toward automated, distributed worm signature detection. *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[5] Automated Worm Fingerprinting

[6] T. Toth, C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. *RAID 2002. LNCS 2516*. pp.274-291. 2002.

[7] *Polymorphic Shellcodes vs. Application IDSs*, NGSEC White Paper, htto:liwuw nesec.com

[8] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software.

Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 05), Feb. 2005.

[9] A. Pasupulati, J . Coit, K. Levitt. S. F. Wu, S.H. Li, J.C. Ku0,K.P. Fan. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. *IEEE/IFIP Network Operation and Management Symposium*, 2004.

[10] J. Newsome, B.Karp and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. *Proceedings. 2005 IEEE Symposium on Security and Privacy*, pp. 226-41, 2005.

[11] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. V. Underduk.Polymorphic shellcode engine using spectrum analysis. http://www.phrack.org/show.php?p=61&a=9.

[12] K2, admmutate. http://www.ktwo.ca/c/ADMmutate-0.8.4.tar.gz.