Preventing Drive-by Download via Inter-Module Communication Monitoring

Chengyu Song Jianwei Zhuge* Xinhui Han Zhiyuan Ye

Key Laboratory of Network and Software Security Assurance (Peking University)
Ministry of Education, China
Institute of Computer Science and Technology, Peking University, China
{songchengyu,zhugejianwei,hanxinhui,yezhiyuan}@icst.pku.edu.cn

ABSTRACT

Drive-by download attack is one of the most severe threats to Internet users. Typically, only visiting a malicious page will result in compromise of the client and infection of malware. By the end of 2008, drive-by download had already become the number one infection vector of malware [5]. The downloaded malware may steal the users' personal identification and password. They may also join botnet to send spams, host phishing site or launch distributed denial of service attacks.

Generally, these attacks rely on successful exploits of the vulnerabilities in web browsers or their plug-ins. Therefore, we proposed an inter-module communication monitoring based technique to detect malicious exploitation of vulnerable components thus preventing the vulnerability being exploited. We have implemented a prototype system that was integrated into the most popular web browser Microsoft Internet Explorer. Experimental results demonstrate that, on our test set, by using vulnerability-based signature, our system could accurately detect all attacks targeting at vulnerabilities in our definitions and produced no false positive. The evaluation also shows the performance penalty is kept low.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*

General Terms

Security

Keywords

Drive-by download, malicious script, inter-module communication, intrusion detection, ActiveX

*Corresponding author.

ASIACCS'10 April 13–16, 2010, Beijing, China.

1. INTRODUCTION

A drive-by download (also mentioned as web-based malware in [32]) is a download which user indirectly authorized but without understanding the consequences or any download that happens without knowledge of the user. As more and more computers are connected to the Internet, this kind of attack has become the most widely use attack vector to deliver malicious code [5, 2, 31]. These malicious codes could be Trojans which steal identification, password of your online game or any other information that can be sold on black market [49]. They may also be bot that used to build botnets [7] to send spams [19], host phishing sites [22], or launch distributed denial of service (DDoS) attacks [21].

A typical drive-by download attack scene has been clearly explained in [30], The general steps may include:

1. Attackers prepare the exploit payloads and malwares. The payloads could be malicious JavaScript, VBScript, browser plug-ins, Adobe Flash or a combination of them. These payloads are then put onto exploit servers and malwares are put onto malware servers.

2. Attackers then embed links pointing to these payloads into legitimate pages [1] by cracking the web sites or buying web visit traffics from the evil or unwitting webmasters.

3. When victims visit these polluted pages, they are redirected to the exploit servers.

4. The browser downloads the exploit payloads and tries to handle them. During the handling procedure, these malicious payloads launch attacks.

5. Once the victims are compromised, the prepared malwares are downloaded and executed.

This new, web-based attack has several advantages over traditional attacks. One advantage is the ability to launch heap-spray attack [27, 8, 38], to increase the success rates of exploitation. But a more deadly one is that, by using this form of attack, it is much easier to bypass most detection mechanisms used by existing protection systems.

Generally, detection mechanisms can be classified into content-based and behavior-based. And protection systems

[1]In this paper, we define a page as all the content (html files, scripts, images, et al.) that will be download automatically by browser during a browsing session.

can be divided into network-based and host-based. Firewall and network-based intrusion detection system (NIDS) are two typical network-based protection systems. Antivirus (AV) software and host-based intrusion detection system (HIDS) are two typical host-based protection systems.

To detect malicious content effectively, the content must have special characteristics distinguishing itself from benign ones. Before drive-by download attack emerged, researchers [41, 29, 28] had found some viable methods to identify malicious network flows by detecting the plain text or light-encoded shellcode within these flows. However, in drive-by download attacks, by utilizing the powerful capacities provided by the script engine of browsers or their plug-ins, attackers can encode their shellcode in much more variable ways during transfer, and then decode them just before the exploitation. Currently, the most widely used encode mechanism is JavaScript obfuscation [14, 32]. And attackers have begun using some more complex mechanisms, such as embedding the malicious script into encrypted Adobe Flash files [47]. Moreover, as the scripting engine getting more and more powerful, attackers may even write a new interpreted language [37] to implement the exploitation.

AV softwares have been fighting against different encoding and morphing mechanisms for many years, and have many mature countermeasures. But most of them still use static scanning strategy to detect malicious content on the web. Although some of them have included script execution engines, comparing with the complex browsers, these engines are so primitive that sometimes they cannot handle the malicious scripts correctly.

In short, current 'unpack' or anti-morphing technologies used by content-based detection mechanisms are not powerful enough to handle these web-based malicious codes. Besides, the scanning strategy is also problematic. Most existing scanners use file or network flow as scanning granularity. However, to help web application developers, most browsers have the ability to let scripts include other scripts. By leveraging this feature, attackers can split one exploit script into several files without affecting the correct functionality. As a result, scanners without a mature reassemble mechanism will be bypassed.

By using behavior-based detection mechanism, the unpacking and reassembling problem stated above could be avoided. But the situation for existing behavior-based protection systems is no better. Firewalls can block illegal traffics, yet from the perspective of firewall, since drive-by download attack is almost the same as legitimate visiting web pages, it has no reason to block this kind of traffic. Even though black lists (e.g. Google Safe Browsing API [18]) can be used to prevent visiting malicious servers, these lists trend to be incomplete and sometimes outdated.

Most HIDSs use system behaviors (e.g. system call invocations) to detect malicious programs. However, as the malicious activities involved in drive-by download attacks (e.g. installing plug-ins, downloading and executing programs) are indistinguishable from legitimate activities at this level, either the false negatives or the false positives will be too high to be acceptable.

The limitation of behavior-based detection mechanism can be attributed to the absence of decisive semantic information from the exploitation scene. That is, the information they gathered are so general that the characteristics of these information are not typical enough for attack detection. This is similar to detecting malicious behaviors happening in the guest operating system at the virtual machine monitor level.

To overcome this limitation, we propose monitoring inter-module communication (IMC) to detect and prevent drive-by download attacks. The motivation is: if we cannot get the decisive semantic information outside the browser, maybe we could get them from the inside. Luckily, after we analyzed the attack scene more carefully, we found the following facts: (1) modern web browsers are modularized, from the basic HTML parsing and rendering engine to the diverse plug-ins, are all implemented as modules; (2) no matter how the malicious content reach the victim and how attackers prepare the exploits, the essence of most exploitations is malicious invocations of functions provided by the vulnerable module; (3) these invocations are from the modules which handle the malicious content; and (4) when the vulnerable function is invoked, the content has usually been deobfuscated. Therefore, we could treat the invocations as communications between modules and by checking these communications, attacks are much easier to detect.

Generally, the proposed detection mechanism works as: (1) monitoring the communications to the vulnerable modules during a browsing session; and (2) checking the communication content to identify known attacks, for improving the detect precision, here we use vulnerability-based signatures [46, 4] instead of traditional attack-based signatures. To verify this idea, we implemented a proof-of-concept system that is integrated into Microsoft Internet Explorer (MSIE) browser. And we tested this prototype system on over one hundred cached drive-by download attack samples.

In summary, the contributions of this paper are:

- We proposed an IMC monitoring based detection mechanism to detect and prevent the drive-by download attacks;
- We described a proof-of-concept implementation of our approach, which can be integrated into the most widely used browser, MSIE;
- We analyzed 19 popular vulnerability reports and generated 37 signatures for those vulnerabilities;
- We evaluated our prototype using 119 cached drive-by download attack samples and the Alexa top ranked sites. The experimental results demonstrate that our system has a high detection effectiveness and low false positive rate, while the performance overhead is kept low.

The rest of this paper is structured as: In Section 2, we introduce the related work and compared our work with them. In Section 3 we present the design of our approach and in Section 4 we describe the implementation of our prototype system. In Section 5 we evaluate the prototype and discuss its limitation and future work in Section 6. At last, we conclude our work in Section 7.

2. RELATED WORK

As drive-by download attack is becoming more and more popular, besides the protection and detection systems mentioned in the introduction section, many other approaches have been proposed to detect, analyze and mitigate this severe threat.

General studies. To get a better understanding of this threat, Provos et al. investigated web-based malware in [32], in which they studied the ways adversaries used to place exploits and the different exploit techniques. In [49] Zhuge et al. gave an empirical study of the black market behind drive-by download attack and reported about 1.49% of the web site returned by search engine is malicious. [31, 9] also studied the malicious web site ratio on Internet.

Server-side detection. To detect injection attacks against benign web pages, Halfond and Orso [16] proposed a static technique to detect SQL injection. In [1] Bandhakavi et al. describes a similar mechanism. Later they extended this mechanism to detect cross site script (XSS) attack [3]. Based on dynamic taint analysis, researchers proposed several methods to detect script injection attack [26, 25, 40] and SQL injection attack [17, 36]. While our work aimed at detecting and preventing drive-by download attacks at the client-side, these works tended to stop the drive-by download attack from the source. However, as some malicious web sites are set up on purpose [49], only detecting server-side exploitations cannot solve the problem once and for all.

Client-side detection. To detect drive-by download attack, by adopting the honeypot idea, Microsoft proposed HoneyMonkey system [45]. On contrary to server-side honeypots that passively wait for attacks, honeypots of this kind (i.e. client honeypot) will actively visit the Internet to trigger web-based attacks. Since legitimate activities are reducing to zero in honeypots, HIDS could be used without producing false positives. However, as most existing high-interaction client honeypot systems [45, 32, 39] still rely on events at system API level, the abnormal behaviors they detected are more likely to be from the malicious activities of downloaded malware, rather than activities directly from the malicious web pages. Hence, if the malware is not downloaded successfully, or the malware does not perform any suspicious activities during the detection time window (e.g. keep silent in virtual machines), a false negative would rise. Since our approach detect the essential exploit behavior, it can detect the attack more precisely than traditional high-interaction client honeypots.

Since high-interaction honeypots are sometimes too heavy, Jose introduced PHoneyC, a low-interaction client honeypot to detect malicious web pages in [24]. Unlike high-interaction client honeypot, PHoneyC uses ActiveX emulation to detect web-based exploits and uses AV scanner to detect known malicious content. Wepawet [20] provides an online service where people can submit suspicious URL for analysis. It uses similar ideas like PHoneyC but is implemented in Java. The disadvantage of these systems is similar to antivirus software, i.e. though they can provide certain level of interaction, today's browsers are too complex to be emulated perfectly. As a result, attackers can use features not easily imitable to detect and bypass these emulators (e.g. innerHTML). Since our approach can be integrated into real browser, we could avoid such problems.

Since most exploits are implemented in script languages, more precisely, in JavaScript [23], detecting and analyzing malicious JavaScript is also gaining more and more attentions. Egele et al. [13] and Ratanaworabhan et al. [33] proposed techniques to identify drive-by download attacks by detecting shellcode in JavaScript variables or heap memory. While our approach is able to detect shellcode-based attacks as well, it also can detect attacks without using shellcode.

Improving reliability of plug-in and browser. As most drive-by download attacks rely on exploiting client-side vulnerabilities, a lot of research has been done on mitigating vulnerabilities or limiting the damages. In Secunia's report [35], it is indicated that many of the vulnerabilities being exploited are from browser plug-ins, especially ActiveX controls. To reduce this kind of vulnerabilities, in [10] Dormann and Plakosh introduced an automated ActiveX control fuzzing system to detect security flaws. To solve this problem from source, Yee et al. [48] and Douceur et al. [11] proposed two new plug-in frameworks named Native Client and Xax. In these frameworks, plug-ins are restricted inside sandboxes like what is done to Java applets but the performance penalty is kept acceptable.

To push these restrictions further, Grier et al. [15] presented OP browser, in which not only different principles are handled in different sandboxed processes, but the scripts and plug-ins used in one principle are also handled in different processes. This browser architecture also emphasizes the important to use centralized, explicit IMC for auditing and security checking. Based on the same idea, Microsoft also developed the Gazelle browser [43]. Although these solutions can provide better security, they are still prototype and most Internet users are tending to use old browsers with which they are familiar. For example, IE6 and IE7 still occupy the biggest portion of the browser market [42]. Comparing with them, our work could provide a more secure browsing experience before the revolution of browser architecture.

Vulnerability-based signature. Vulnerability-based signatures have been use in the Shield system [44] introduced by Wang et al. Later, this work was extended to prevent web-based exploits in [34]. Our work, while also aims at preventing web-based exploits, differs from BrowserShield in the interception level. BrowserShield instruments HTML and JavaScript content of a web page but our system intercepts the communications made between browser modules. Since the malicious content is not necessarily in the HTML files or JavaScript files, we believe our architecture could detect more attack forms.

In [4], besides giving the formal definition of vulnerability-based signature, Brumley et al. described an automated, data flow analysis based way for generating vulnerability signatures. And in [6], Cui et al. proposed an automatic way to generate vulnerability-based signatures for unknown vulnerabilities. Therefore, though in our prototype system, the signatures are manually generated, we can use these proposed methods to generate vulnerability-based signatures in automated manner, to increase the detection scope.

3. SYSTEM DESIGN

In this section we describe the design of our approach. Firstly, we discuss the threat model and system boundaries that guide our design. Then we give an overview of our approach and describe the functionality of each component.

3.1 Threat model and system boundaries

We designed our system to handle drive-by download attacks. We assume attackers could have the complete control over the web server. Therefore, they can prepare any kind of exploit that could be delivered through web browsing. We also assume the exploits may arrive at the browser by any means, in any format and may target at any part of the browser, including its plug-ins.

We only consider attacks that originate from web and target at web browsers and their plug-ins. We also limit our system only to detect attacks that require invocation of inter-module communication.

3.2 System Overview

Our system consists of two main components, a *Monitor* which monitors the IMC and generates security events; and a *Detector* that identifies attacks targeting at known vulnerabilities from the event trace..

Although IMC is also a kind of local function invocation, in many browsers, it is different from calling an application program interface (API). An API is permanent, always available to be called. But most component model (COM, XPCOM) used in web browsers use objects that are created at demand, and destroyed when no longer needed to provide functionalities. For example, when visiting Youtube, an Adobe Flash Player object is created to play the video; and as soon as you leave that page, this object is freed thus not available for reuse. From this perspective, IMC is more similar to network communication where communication can have a session. Therefore, in our system, we define three kinds of events to stand for the whole procedure of IMC:

Object creation. A creation of a component object indicates the beginning of a new communication session.

Method invocation. Invocations of methods constitute the main part of the communication.

Object free. The free of a component object indicates the end of the session.

3.3 The Monitor

The IMC Monitor is in charge of generating the three kinds of events. In our approach, it is an abstract component that can be implemented in different ways.

If integrated into current browsers (e.g. MSIE, Firefox), where implicit IMC is used, the Monitor can be implemented as a plug-in that will be loaded at the startup time. This plug-in then intercepts the system calls for object creation to generate corresponding events. However, after the component objects are created, the IMC is made implicitly. Therefore, method invocation events and object free events must be generated differently. A general method is to hook every method the created object offers, including the object's destructor. After that, when a hooked function is called, a method invocation event is generated. And whenever the destructor is called, an object free event is generated.

In future browser architectures like the OP browser, IMC must be invoked explicitly by calling the IMC API provided by the browser kernel, and the kernel will monitor and audit these invocations. Under this circumstance, the Monitor can simply be registered as a handler for the three kinds of events. Or it could be implemented as a parser that generates these events from the audit log.

The Monitor can also be built into a low-interaction client honeypot. For example, by modifying the ActiveX emulation component already exists in PHoneyC.

3.4 The Detector

The Detector is responsible for detecting known attacks by matching the generated IMC events with signatures in its vulnerability definitions. To improve the detection precision, we use the vulnerability-based signature to detect attacks.

In [4], a vulnerability signature is formally defined as a matching function which, for an input x returns either EXPLOIT or BENIGN without running the original vulnerable program P. The authors also introduced three main signature representation classes, *the Turing machine signatures*, *the symbolic constraint signatures* and *the regular expression signatures*. The Turing machine signatures are precise but may take an unbounded time for matching. Matching regular expression signatures are efficient but the language itself has fundamental limitations that may tamper the precision. Therefore, in our system, the Detector uses the symbolic constraint signatures for detection.

One modification here is the definition of input x. Some of the vulnerabilities require several method invocations before the components can be successfully compromised. For example, exploiting MS08-041 requires first setting the `SnapshotPath`, then setting the `CompressedPath`. Hereby, we need to track the session state to indicate what input has already been received by the vulnerable object. Because one object may contain more than one vulnerability (e.g. CVE-2007-4816), the session state must be tracked separately for each signature.

In consideration of these facts, we formally define a signature used in our system as a deterministic finite state automaton (DFA)

$$S = (Q, \Sigma, \delta, q_0, F)$$

Each state $q \in Q$ indicates the current state of the session, and q_0 is the initial state. When an object is created, the instantiated signatures for this object are initialized with q_0. Σ represents the possible input symbol for the object and $\Sigma*$ is all the possible input x for that object. δ is a symbolic constraint resolver that checks if the input $x \in \Sigma*$ at state q_a satisfies the symbolic constraint on q_a. If it does, the signature transits to the new state q_b; otherwise it remains in current state. There are two final states $f_1, f_2 \in F$. f_1 represents EXPLOIT and f_2 represents BENIGN. As soon as the state transits to f_1, an attack is detected. And for the reason that each component object may be exploited at any time during its life time, f_2 state can only be reached when the object free event is received.

4. PROTOTYPE IMPLEMENTATION

In this section, we describe the implementation of our prototype system in detail. In the first part, we provide a short overview of the Microsoft Internet Explorer browser and its plug-in system ActiveX. Then we present the implementation of COMSniffer (the Monitor) and MwDetector (the Detector).

4.1 MSIE and ActiveX

MSIE is a series of graphic web browsers developed by Microsoft, and included as part of the Microsoft Windows since 1995. It has been the most widely used web browser since 1999. Although the latest version IE8 has many security improvements to mitigate drive-by download, XSS and phishing attacks, the most widely used version is still IE6. MSIE uses a componentized architecture built with the Component Object Model (COM) , which is also the basic of its plug-in framework – ActiveX. For client-side scripting, MSIE supports JavaScript and VBScript by default. It can also support CLI-languages through Silverlight.

The COM technology is introduced by Microsoft in 1993 as an extension of its OLE model. The key idea is to implement a component developing model that is Object Oriented (OO). Thereby, compare with dynamic link library (DLL), COM components are more similar to classes in OO languages. The functionalities of a component are provided by objects. Each object has its own private data, and offers the functionalities through different methods of the interfaces it implements. And Components also support reuse. The most significant difference is that COM objects' interface is binary compatible, thus can be implemented in any languages on any platform. Another important feature of COM is process transparency. That is, when using a COM component, the user does not need to care whether the component is implemented as a DLL inside its process or a separate process on local system, or even a process on remote server. Each COM component is identified by a global unique id called *classid* (CLSID).

Interface is one of the core concepts in COM. One interface represents a certain set of functionalities (e.g. IHTMLDocument) provided by an object. Object must implement at least one interface, but it may implement several interfaces as well. Similar to COM components, every interface has a global unique id called *interface id* (IID). The basic and most important interface is *IUnknown* interface. Every interface must 'inherit' this interface and implement the three methods of this interface: `AddRef`, `Release` and `QueryInterface`.

The basic steps to utilize a COM component are: (1) create a component object; (2) use the object's QueryInterface method to query the interface that contains the demanded method; and (3) call the method.

To manage large amount of COM components, Microsoft also introduced the concept of *COM library*. COM library provides interfaces to register, query, create and remove COM components. It also provides utility functions to handle common COM data structures and to manage memory usage.

ActiveX framework is the plug-in system for MSIE introduced in 1996 as a competitor to Java applet. A plug-in implemented using ActiveX is usually called an ActiveX control. Since ActiveX control is also a kind of COM component, it can provide much richer abilities than applet. But on the other hand, it also increases security risks. These security risks exacerbate after MSIE allows client-side scripts to interact with ActiveX controls.

4.2 COMSniffer

In this section, we first describe how COMSniffer generates the three kinds of IMC events. Then we discuss the performance optimization and some implementation details.

4.2.1 Generating object creation events

An object creation event for COM object contains two piece of information: the object's CLSID and the object's address. The CLSID helps the Detector decide what signatures should be used for this session. For the reason that a COM object is used directly as an interface pointer, in our prototype system, we use the object's address as the ID of an object. Since an object represents a session, this address is also used as the ID of the session.

To generate the object creation events, we need to understand the procedure of creating a COM object. Most

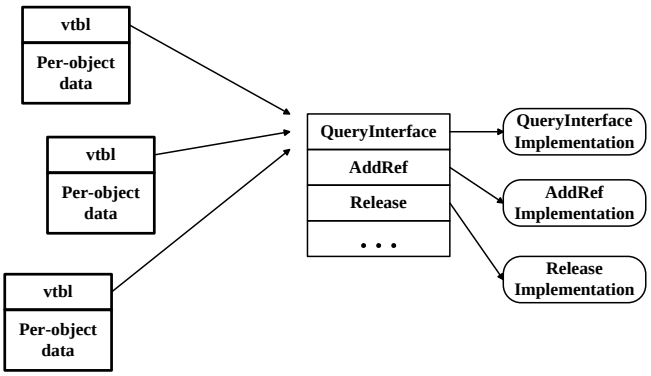

Figure 1: Binary Format of COM Interface

COM objects used by MSIE are created in two ways. The first one is to call the `CoCreateInstanceEx` API [2] provided by the COM library, with the required CLSID and IID. If succeeds, this API directly returns the required object. Because CLSID and created object's address is passed either as an input parameter or as an output parameter, to monitor objects created through this way, we intercept the `CoCreateInstanceEx` API.

The second way is to call the `CoGetClassObject` API with the required CLSID. Unlike `CoCreateInstanceEx`, this API does not return a object of that CLSID, but a class factory object (object that implements *IClassFactory* interface). To create the 'real' object, the caller then calls the `CreateInstance` method of the class factory with the demanded IID. As a result, monitoring object created in this way is more complicated. Not only because the CLSID information and object's address is involved in two function calls, but also because the later function is not a system API. To solve this problem, we first intercept the `CoGetClassObject` API. When a class factory object is created, we hook its `CreateInstance` method and its CLSID is recorded. Then, when the `CreateInstance` method is invoked, we could obtain the object's address. The next step is to correlate the CLSID and the object's address. In our system, this is done by mapping the CLSID to the class factory object's virtual function table. This is feasible because the binary structure of COM object (Fig. 1) is similar to a C++ object. The first element of an object is a pointer that pointed to the interface's virtual function table (vtbl). And all objects from the same component, with the same interface use the same vtbl. So, when a class factory object is created, we save the CLSID - IClassFactory vtbl pair. And when the hooked `CreateInstance` method is called, we obtain the vtbl address through `This` pointer [3], get the corresponding CLSID, and generate a creation event.

4.2.2 Generating method invocation events

A method invocation event contains three pieces of information: the object address, the method name and the parameters. The method name, together with the parameters, forms the input to be checked.

The ordinary way to monitor function invocation is to in-

[2]The `CoCreateInstance` API is also implemented by this function.

[3]Like C++ and other OO languages, the first parameter of any COM interface's mehtod is a 'this' pointer.

tercept the very function, by adding a hook, setting a debug break point or et al. However, since the number of COM components and their interfaces is extremely large, writing callback functions for every method is too expensive. Luckily, when creating the ActiveX framework, Microsoft also defined the standard interface through which client-side scripts can interact with the ActiveX controls. This interface is also the interface every ActiveX control must implement and the only way to interact with scripts. Its name is *IDispatch* interface [4]. The two key methods of this interface are `GetIDsOfNames` and `Invoke`. When a script wants to use an ActiveX control, after creating the object via `<object>` tag or `ActiveXObject` function, it cannot directly call the method. Rather, the scripting engine (e.g. JScript) will query the IDispatch interface of that object, and calls the `GetIDsOfNames` method to check if that component has this method. If it does, `GetIDsOfNames` will return the corresponding dispatch ID (DISPID). The engine then calls the `Invoke` method to finish the method invocation, using the DISPID.

Therefore, to generate method invocation events, after a successful creation of a COM object, we check if this component supports IDispatch interface. If it does, we hook the `GetIDsOfNames` and `Invoke` method of this object. When the hooked `GetIDsOfNames` is called, we save the method name - DISPID pair. And when the hooked `Invoke` method is called, we obtain the corresponding method name of the given DISPID, gather the parameters information and generate a method invocation event.

To prevent the exploit, we make COMSniffer work in inline mode. When in this mode, COMSniffer will wait until the Detector returns the checking result. If an attack is detected, COMSniffer reports the attack and then *prevents* the original method invocation by directly returning an error code.

4.2.3 Generating object free event

Generating object free events is easier. COM standard defines: when the calling of `Release` method returns 0, this object is going to be released. Hence, we monitor the `Release` method by hooking this method of every COM object on their creation. And when the original function returns 0, an object free event is generated.

4.2.4 Performance Optimization

Since MSIE is built around COM and most of its components support IDispatch, COMSniffer may generate too much noise information. To reduce these noises, we only monitor CLSIDs that exist in our vulnerability definitions. This optimization will not affect the detection because our Detector only detects attacks targeting known vulnerabilities.

4.2.5 Implementation

Although most security plug-ins for MSIE are implemented as browser helper objects (BHO), COMSniffer is not. This is because by the time BHOs are loaded, some of the core components object of MSIE have already been created, thus cannot be monitored by COMSniffer. So we implement it as a DLL that will be loaded by MSIE immediately after the process is created. And once loaded, it will automatically start monitoring.

[4]Microsoft later extended this interface with *IDispatchEx* interface, but for our prototype, IDispatch is enough.

4.3 MwDetector

In this section, we describe the implementation of MwDetector, the Detector in our prototype system.

4.3.1 Signatures

In MwDetector, we store DFA definitions of signatures in description files, one signature per file. The file content is constructed as follows.

- The first line of a definition file is the CLSID of the vulnerable component, and the rest content is divided into one or more blocks, every block is separated by a new line.
- Each block describes one part of the constraint.
- The first line of the block is the original state number. The initial state uses number 1, and the attack state uses number 0. The rest numbers (positive) are free to use.
- The second line is the target state number. The numbering rule is the same as above.
- The third line is the method name, which according to ActiveX standard, is case insensitive.
- The rest of the block is the expressions of symbolic constraint associated with the original state. Since the resolver used in our prototype system is Yices [12], therefore these expressions are written in Yices Input Language.

At runtime, MwDetector uses a *signature manager* to manage the signatures in its vulnerability definitions.

4.3.2 Session manager

In MwDetector, the three kinds of IMC events are handled by *session manager*. When an object creation event is received, the session manager creates a new session object. Then it uses the object's CLSID to query the signature manager for *matchers* (discussed below) that belong to this CLSID, and associates these matchers with the created session. After that, this session object is put into the living session list.

When a method invocation event is arrived, the session manager uses the object address to find the corresponding session in the session list. Once found, the session manager feeds all the associated matchers with the method name and parameters passed in. If any matcher enters the EXPLOIT state, the session manager raises an attack alert. The alert information is then returned to COMSniffer for further processing.

To handle the object (COM object) free event, the session manager first finds the session object the same way as above, and removes the session object from the live session list. Then, the session manager frees all the associated matchers and the session object itself.

```
typedef struct tagDISPPARAMS
    {
    VARIANTARG *rgvarg;
    DISPID *rgdispidNamedArgs;
    UINT cArgs;
    UINT cNamedArgs;
    }   DISPPARAMS;
```

Figure 2: Structure of DISPPARAMS

4.3.3 Matcher

In MwDetector, a *matcher* is an instance of a signature. When MwDetector is loaded, the signature manager will load all the signatures by parsing the definition files. When the session manager queries for the matchers, the signature manager instantiates signatures registered for handling events of this CLSID and returns the matchers to the session manager.

There could be several kinds of matcher for different kinds of input data. But every matcher class has to implement a transition method. This method is called by the session manager on the receiving of method invocation events. For COM matchers, the transition method works as follows.

Firstly, the parameters are parsed into symbolic expressions. The parameters used to invoke a method through the IDispatch interface are stored in a `DISPPARAMS` structure (Figure 2). The `cArgs` member stores the number of unnamed arguments, while the `cNamedArgs` member stores the number of named ones. The information of each argument is store in `rgvarg` and `rgdispidNamedArgs` respectively. Since most scripting languages are weakly-typed, to support interaction with these languages, each unnamed arguments is stored in a `VARIANTARG` structure. The first member of this structure indicates the type of the argument. Commonly used types include strings (`VT_BSTR`), integers (`VT_I`k, `VT_UI`k, $k = 1, 2, 4, 8$), objects (`VT_DISPATCH`) and variables [5] (`VT_VARIANT`). The parser will generate expression according to the argument's type. For example, a string parameter will be expressed as:

```
  (define arg#_index#::int)
(assert (= arg#_index# value))
```

Then the matcher locates the transition constraint according to current state and the name of the invoked method. Then the matcher calls Yices to check whether the input is consistent with the constraint. If it is, the state of the session transits to the target state. Else, the state remains in current state. Since there could be several matchers for one session, the session state is stored inside each matcher to avoid interference.

5. EVALUATION

In this section, we discuss how our prototype is evaluated and the experimental results. We first introduce the test environment we used to carry out the rest evaluations. Then we describe the three experiments. The first one evaluates the detection effectiveness, i.e. false negative rates. The second one evaluates the false positive rate. And the last one measures the performance overhead.

5.1 Drive-by download attack replaying system

[5]The type of this argument is unsure.

Drive-by download attack has been a hot topic for several years, however, up till now, a standard test base for measuring the effectiveness still has not emerged. In this section, we give a brief introduction of our test environment which is able to reliably replay the drive-by download attacks.

After drive-by download has become a major threat to the Internet users, security research groups and companies have built several system to detect these attacks. Some of them will publish a list of recently detected malicious pages. These pages should be used to evaluate new mitigation approaches. Unfortunately, most of these malicious pages only live for a short time before they are cleaned up. After that, they are no longer capable to be used for evaluations.

To overcome this limitation, most researchers would store a local copy of the malicious page. But a drive-by download attack usually involve tens of files that distributed over several web servers, by using `<script>`, `<frame>` and `<iframe>` tags. If only one file is visited during the evaluation, the exploit may not be triggered correctly [13]. One would suggest modifying the `src` attributes of those tags. For static tags, this would work. But many malicious pages will dynamically create those tags, and the pages themselves are heavily obfuscated or even encrypted using the correct URL [13], hence makes this approach infeasible.

To solve this problem, we build a proxy based replaying system. Once we find a malicious page, we cache in local every web content (HTML documents, scripts, images, binary files, et al.) involved in the attack. Thereafter, when we want to revisit those pages to evaluate a system, we can simply set the system to use that proxy, and then directly visit the original URL, the replaying system will response with all cached web content in the attack scene, therefore restoring the original drive-by download attack scene.

5.2 ActiveX emulator

Most drive-by download attacks will try to exploit multiple vulnerabilities to improve the success exploitation ratio. Hence, to build a good detection system, it is necessary to install as much vulnerable ActiveX controls as possible. Unfortunately, not all controls are compatible. For example, during the development of our system, we found when Microsoft Access Snapshot Viewer (CVE-2008-2463) is in use, other vulnerabilities cannot be exploited within the same process. To solve this problem, in our detection environment, we create an universal ActiveX control that can be instantiated as any ActiveX component. As a result, whenever a malicious page attempts to utilize an ActiveX control that does not exist in our environment and invoke its vulnerable method, this attempt will succeed and the attack will be detected by MwDetector. To trigger attacks that rely on results of method invocations (e.g. the version of RealPlayer), our ActiveX emulator is able to return this demanded information (not a dummy component).

Since most drive-by download attacks rely on ActiveX controls, different kind of emulators have also been implemented in previous works [24, 20]. However, most of these emulators are implemented in scripting language, the same level as the malicious scripts. Therefore, they are not completely invisible to drive-by download exploits. On the contrary, the simulation in our system is at COM level, a level below the script engine. This means, it is not easily detectable for malicious scripts.

5.3 Detection effectiveness

To measure the detection effectiveness of our prototype, we first evaluated it on 119 in-the-wild drive-by download attack samples (1010 html and script files) that were cached by our replay system. All samples were detected by our own high-interaction honeypot. These samples are visited in a batch mode, each given 2 minutes to process.

The tested system is integrated into IE6 browser on a clean installed Windows XP SP2 with no more patches. In addition, Adobe Flash Player 9.0.47.0 is installed. We use this configuration because it is a typical vulnerable environment for web browsing. However, in theory our prototype can also be integrated with other MSIE versions on different Windows versions.

For this evaluation, we manually generated 37 signatures from 19 vulnerability reports (listed in Appendix A). Since all these signatures are extracted from vulnerable components, there is no training samples used in this evaluation.

The result is, with the help of the ActiveX emulator, our detection system successfully detected 895 exploit instances (one sample may contain several attacks targeting the same vulnerability) from 99 samples (Figure 3). This yields an initial detection effectiveness of 83%, i.e. a false negative ratio of 17%. To understand why our system did not report any attack on the remaining 20 pages, we reanalyzed them with our high interaction honeypot. The reexamination revealed that none these samples are active. This is due to cache problem of our replay system, more precisely, not all files involved in these samples are successfully cached. After excluding these inactive samples from our dataset, we compute a detection rate of 100%.

Besides the detection effectiveness, this experiment also showed two interesting results. The first one is, by using the emulator, in addition to the vulnerabilities that do exist in test environment (CVE-2006-0003), our system also detected attacks tried to exploit other 14 vulnerabilities. The other one is, though vulnerabilities which can be exploited without shellcode (CVE-2006-0003, CVE-2008-2463, CVE-2007-4105, CVE-2008-6442) is relatively rare, the number of attacks that target them is much larger than those aiming at exploiting vulnerabilities of other kinds. This means, existing memory protection mechanisms like data execution prevention (DEP), address space layout randomization (ASLR) and shellcode-based detection systems [13, 33] are not sufficient to protect users from drive-by download attacks.

From the perspective of detecting malicious pages, the detection rate of our system is perfect. However, since most of the cached samples contain more than one exploit, this result cannot fully represent the detection effectiveness of all detectable exploits. Therefore, we further chose 5 samples with disparate attack payloads [6] and manually analyzed them to find out all the exploits. Then we compared this result with attacks detected by our system. The result is shown in Table 1. The detection rate on this small dataset decreased to 52%. The reason for this decrease is that those undetected exploits are aiming at vulnerabilities that are not included in our definitions yet. This is an unavoidable weakness of all signature-based detection systems. But our system detected all attacks targeting at vulnerabilities in our definitions.

[6] Although the initial page of every sample is different, the exploit files involved could be similar or even the same.

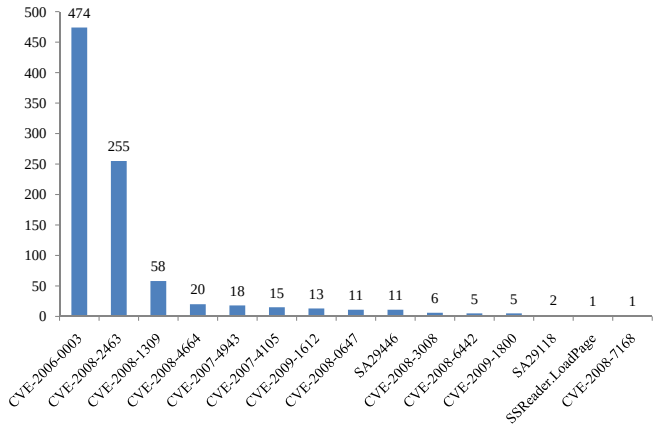

Figure 3: Exploit instances detected on the 119 cached samples.

Table 1: Exploit instances detected on the manually analyzed samples

Sample ID	Exploits found	Exploits detected
1	6	6
6	3	1
24	9	6
26	13	2
67	3	3

5.4 False positive evaluation

In the context of our system, a false positive is a page that is detected as containing at least one exploit, but in fact does not. To evaluate the likelihood of false positives, we drive our prototype to visit the home page of the top 100 sites of Alexa global ranking and of the top 100 site within China. All these pages are known to be benign. And we consider them as a reasonable test set that represents the most commonly visit content of Internet users.

On this dataset, our prototype did not produce any false positive. This 'perfect' result is expected. There are two reasons. The first one is, the detection is based on vulnerability-based signature, and most vulnerabilities we met are relatively simple (e.g. stack overflow), hence the false positive and false negative should be naturally low. And the second one is, most of these commonly visited sites did not instantiate any of the vulnerable ActiveX controls, or did not invoke the vulnerable methods that exist in our vulnerability definitions.

5.5 Performance

Though the detector can be fed by a log parser (i.e. works offline), our prototype is implemented in online mode. That is, the detection is performed at the same time when the MSIE handles the page. Thence, we evaluated the performance penalty caused by our prototype in this section.

This experiment is carried out on the Alexa top 100 sites within China. We first uninstalled our system and drove IE6 to visit the dataset and recorded the time used (wall-clock time). This is done by installing a BHO that record the time elapsed between the `BeforeNavigate2` event and the `DocumentComplete` event from the root frame. After that,

Table 2: Page load times (sec) with and without our detection system.

	Native MSIE	With detection
Total time	410	471
Average time	4.10	4.71
Average Overhead	0	**0.61**
Factor	1	**1.15**

we reinstalled our detection system on the same machine (without the ActiveX emulator, works as a prevention system) and ran the same test again. The local cache, history information and are cookies cleared before every test.

The evaluated system was of the same configuration as previous experiments and was installed on a virtual machine with Intel Core 2 Duo 2.4G CPU (one CPU mode) and 256MB of main memory. An ADSL of bandwidth 2Mb/s is used to visit the Internet.

The result is presented in Table 2. On average, native MSIE uses 4.1 seconds to load a page. This time includes downloading all the files within the principle, parsing and rendering it and executing all dynamic content. After integrated our system, the average load time increased to 4.71 seconds, i.e. an overhead of about 15%. Although most Internet user would expect the browsing experience as fast as possible, comparing with the protection provided by our system, we believe this penalty is a fair tradeoff.

6. DISCUSSION AND FUTURE WORK

Our prototype system has several limitations. First, it only detected attacks using explicit malicious content. That is, the malicious content is directly passed to the vulnerable component in the arguments. However, some attacks will use implicit malicious content. For example, some pages may instantiate an Adobe Flash Player control and pass it a URL leading a malicious flash file (CVE-2007-0071). Other examples include malicious PDF files (CVE-2007-5659), malformatted picture files (CVE-2007-0038) and et al. To detect such attacks, we can extend our system to (1) monitor the uniform resource identifier (URI) passed to the vulnerable component; (2) intercept the downloading procedure; and (3) once a monitored URI finished downloading, before the content is passed to the component, check if this content contains any exploit.

The second one is the limitation of all signature-based detection systems: they can only detect known attacks. Therefore, our system cannot detect new, zero-day attacks, or attacks aiming vulnerabilities not in our definitions. There is no ideal solution for this limitation now. But we still could audit the IMC and add some heuristic rules to detect suspicious behaviors (e.g. string argument that is too long). After that, we can manually or automatically analyze these suspicious samples to see if they're false positives or new, unknown attacks.

The third one is that the signatures we used now are manually generated. This work is time consuming and error-prone. However, since the symbolic constraint is general and has widely used in vulnerability analysis and mining systems, we plan to develop our automated signature generating system in the future. Besides, the vulnerabilities we analyzed now are either collected from published database (e.g. CVE) or from captured attack samples. Once we have an automatic signature generating system, we can further modify it to dig common vulnerabilities (e.g. buffer overflows) in ActiveX controls.

7. CONCLUSION

Drive-by download attack is one of the most severe threats to Internet users, and is now the number one source of infecting malware. Most drive-by download attacks rely on the success of the client-side exploits. Most of these exploits are launched against the browser components that handle the malicious content or other vulnerable browser plug-ins. Therefore, in this paper, we proposed a novel approach to detect and prevent such exploits by monitoring inter-module communications between browser components and its plug-ins. By using vulnerability-based signatures, our system is able to detect the malicious exploitation of a vulnerable component before the vulnerability is exploited.

To demonstrate the feasibility of our approach, we developed a prototype system that is integrated into Microsoft Internet Explorer. Experiments on this prototype system showed that, on our test set, our system has a detection rate of 100% on exploit instances attacking vulnerabilities in our definitions. And the false positive ratio is kept 0. And this protection only introduced a performance overhead of 15%.

Because our solution is conceptually generic, it can also be integrated into other browsers like Mozilla Firefox or even new browser architectures like OP browser. By integrating into low interaction client honeypot like PHoneyC, it also can work like an intrusion detection system to identify malicious web pages.

8. ACKNOWLEDGMENT

This work is supported by the Research Fund for the Doctoral Program of Higher Education of China under Grant No.200800011019.

We would like to thank all the anonymous reviewers for their insightful comments and feedback. We would like to thank Tao Wei, Zhiyin Liang, Xiaorui Gong, Jinpeng Guo and Jinhui Zhong for their comments on our research.

9. REFERENCES

[1] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. Candid: preventing sql injection attacks using dynamic candidate evaluations. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 12–24, New York, NY, USA, 2007. ACM.

[2] L. Beijing Rising International Software Co. Internet security report for china mainland, 2009 h1. http://it.rising.com.cn/new2008/News/NewsInfo/2009-07-21/1248160663d53890.shtml, November 2008.

[3] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, pages 23–43. Springer Berlin / Heidelberg, 2008.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of Vulnerability-Based signatures. In *Proceedings of the*

2006 IEEE Symposium on Security and Privacy, pages 2–16. IEEE Computer Society, 2006.

[5] M. Cruz. Most abused infection vector. http://blog.trendmicro.com/most-abused-infection-vector/, December 2008.

[6] W. Cui, M. Peinado, H. J. Wang, and M. E. Locasto. Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 252–266, Washington, DC, USA, 2007. IEEE Computer Society.

[7] D. Dagon, G. Gu, C. P. Lee, and W. Lee. A taxonomy of botnet structures. *Computer Security Applications Conference, Annual*, 0:325–339, 2007.

[8] M. Daniel, J. Honoroff, and C. Miller. Engineering heap overflow exploits with javascript. In *WOOT '08: Proceedings of the 2nd USENIX Workshop on Offensive Technologies*, July 2008.

[9] O. Day, B. Palmen, and R. Greenstadt. Reinterpreting the DisclosureDebate for web infections. In *Managing Information Risk and the Economics of Security*, pages 1–19. Springer US, 2009.

[10] W. Dormann and D. Plakosh. Vulnerability detection in activex controls through automated fuzz test. http://www.cert.org/archive/pdf/dranzer.pdf, 2008.

[11] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI '08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, December 2008.

[12] B. Dutertre and L. D. Moura. The yices smt solver. Technical report, SRI International, 2006.

[13] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *DIMVA '09: Proceedings of the 6th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, July 2009.

[14] B. Feinstein and D. Peck. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. http://mirror.fpux.com/HackerCons/BlackHat_2007/BlackHat/Presentations/Feinstien_and_Peck/Whitepaper/bh-usa-07-feinstien_and_peck-WP.pdf, 2007.

[15] C. Grier, S. Tang, and S. T. King. Secure web browsing with the op web browser. *Security and Privacy, IEEE Symposium on*, 0:402–416, 2008.

[16] W. G. J. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 174–183, New York, NY, USA, 2005. ACM.

[17] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185, New York, NY, USA, 2006. ACM.

[18] G. Inc. Google safe browsing api. http://code.google.com/apis/safebrowsing/.

[19] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: an empirical analysis of spam marketing conversion. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 3–14, New York, NY, USA, 2008. ACM.

[20] U. C. S. Lab. Wepawet. http://wepawet.iseclab.org/.

[21] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring internet denial-of-service activity. *ACM Trans. Comput. Syst.*, 24(2):115–139, 2006.

[22] T. Moore and R. Clayton. An empirical analysis of the current state of phishing attack and defence. In *WEIS '07: Proceedings of the Sixth Workshop on the Economics of Information Security*, 2007.

[23] Mozilla. Spidermonkey (javascript-c) engine. http://www.mozilla.org/js/spidermonkey/, 2009.

[24] J. Nazario. Phoneyc: A virtual client honeypot. In *LEET '09: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*. USENIX Association, 2009.

[25] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Security and Privacy in the Age of Ubiquitous Computing*, volume 181 of *IFIP International Federation for Information Processing*, pages 295–307. Springer Boston, 2005.

[26] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 124–145. Springer Berlin / Heidelberg, 2006.

[27] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[28] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Recent Advances in Intrusion Detection*, volume 4637 of *Lecture Notes in Computer Science*, pages 87–106. Springer Berlin / Heidelberg, 2007.

[29] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorphic shellcode detection using emulation. *Journal in Computer Virology*, 2(4):257–274, February 2007.

[30] T. H. Project. Know your enemy: Malicious web servers, August 2007.

[31] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Security '08: Proceedings of the 17th Usenix Security Symposium*, pages 1–15, Berkeley, CA, USA, 2008. USENIX Association.

[32] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 4–4, Berkeley, CA, USA, 2007. USENIX Association.

[33] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection

attacks. In *Security '09: Proceedings of the 18th USENIX Security Symposium*, 2009.

[34] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Trans. Web*, 1(3):11, 2007.

[35] Secunia. 2008 report. http://secunia.com/gfx/Secunia2008Report.pdf, 2008.

[36] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS '09: Proceedings of the 16th Annual Network & Distributed System Security Symposium*, San Diego, CA, Februry 2009.

[37] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. *Security and Privacy, IEEE Symposium on*, 0:94–109, 2009.

[38] A. Sotirov. Heap feng shui in javascript. http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html, 2008.

[39] R. Steenson and C. Seifert. Capture-hpc client honeypot / honeyclient. https://projects.honeynet.org/capture-hpc/.

[40] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM.

[41] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract pay load execution. In *Recent Advances in Intrusion Detection*, volume 2516 of *Lecture Notes in Computer Science*, pages 274–291. Springer Berlin / Heidelberg, 2002.

[42] W3Counter. Global web stats. 2009.

[43] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *Security '09: 19th USENIX Security Symposium*, August 2009.

[44] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. *SIGCOMM Comput. Commun. Rev.*, 34(4):193–204, 2004.

[45] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2006*, San Diego, California, USA, 2006.

[46] Y.-M. Wang, R. Roussev, C. Verbowski, A. Johnson, M.-W. Wu, Y. Huang, and S.-Y. Kuo. Gatekeeper: Monitoring auto-start extensibility points (aseps) for spyware management. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 33–46, Berkeley, CA, USA, 2004. USENIX Association.

[47] J. Wolf. Heap spraying with actionscript. http://blog.fireeye.com/research/2009/07/actionscript_heap_spray.html, 2009.

[48] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[49] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou. Studying malicious websites and the underground economyon the chinese web. In *Managing Information Risk and the Economics of Security*, pages 1–20. Springer US, 2009.

APPENDIX

A. GENERATED SIGNATURES

For the evaluation, we generated 37 signatures from 19 vulnerability reports (listed in Table 3). These signatures are manually generated by analyzing (e.g. debugging) the vulnerable components and identifying the symbolic constraint required for an exploit. Note, consisting with the signatures does not imply the attack will succeed. For example, the local string buffer used by vulnerable method Register from SSReader Pdg2 ActiveX control (CVE-2007-5892) is 256 bytes long, and any string passed in longer than this will be reported as an attack; but too overwrite the return address on stack, the string passed in should contain least 264 characters.

Table 3: Signatures included in prototype implementation

Component Nam	Vulneratibility
MS MDAC RDS.Dataspace	CVE-2006-0003
Baidu Soba Search Bar	CVE-2007-4105
Storm Player MPS	CVE-2007-4816
Storm Player SPARSER	CVE-2007-4943
RealNetworks RealPlayer	CVE-2007-5601
SSReader	CVE-2007-5807
Thunder PPlayer	CVE-2007-6144
Ourgame HanGamePluginCn	CVE-2008-0647
RealNetworks RealPlayer	CVE-2008-1309
MS Access Snapshot Viewer	CVE-2008-2463
MS Windows Media Encoder	CVE-2008-3008
QVOD Player	CVE-2008-4664
Sina UC Dloader	CVE-2008-6442
UUSee Player	CVE-2008-7168
MS Office Spreadsheet	CVE-2009-1136
Chinagames iGame	CVE-2009-1800
Ourgame GLWorld	SA29118
Ourgame GLWEBAVT	SA29446
SSReader	NO-CVE(LoadPage)