通常,网页木马的攻击步骤可分为 3 步.

(1) 获取和准备攻击脚本和 Shellcode.受害者在访问着陆页时,该页面将利用放置 script,iframe 或跳转的方式从恶意软件分发站点获取攻击脚本和 Shellcode.着陆页面代码和攻击脚本通常会被混淆,以逃避杀毒软件的检测.恶意脚本为了增加攻击的成功率,可能会检测用户的系统环境、浏览器、插件等信息,并可能同时对多种漏洞进行攻击.

(2) 利用漏洞.攻击脚本利用受害者系统中的漏洞,注入 Shellcode,将控制流跳转到 Shellcode 上启动恶意软件的下载和安装,完成攻击者的目标.

(3) 返回攻击脚本,下载和安装恶意软件.从远程恶意软件分发站点下载恶意软件,并在受害者不知情的情况下安装.被安装的恶意软件可能是病毒、后门程序、木马、广告程序和间谍程序等.

为了对用户发动攻击,攻击者可以自己搭建一个站点部署恶意代码诱骗用户访问,但根据文献[19]的研究,现实世界中绝大部分攻击为了扩大攻击范围,都会通过各种方法将攻击脚本嵌入到一个正规页面,或使对正规页面的访问重定向到恶意软件分发站点.要达到这个目的,可以使用的手段有:利用网站服务器漏洞获取控制权,篡改网站;利用网页应用的漏洞进行 XSS,SQL 注入等攻击,使其他用户在浏览时或服务器在进行安全相关的操作时执行攻击者提供的内容;针对引用和混合其他来源服务的页面,例如第三方广告、插件和代码库,攻击者可利用这些外部来源的内容发动攻击.

2.2 例 子

Trojan:JS/Tadtruss.A(https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Trojan%3AJS%2FTadtruss.A)是一种 JavaScript 恶意代码,使用户浏览器跳转到恶意软件发布站点,图 2 所示为其样本示例.这段代码被放置在某个站点上,并进行了代码混淆处理,以隐藏真实意图.*tol* 函数将传进的包含混淆后代码的字符串解密并用 *eval* 函数动态执行.图中最后一行代码中传入 *tol* 函数的字符串,解开后是一段用 window.location 跳转到域名为 bestangelsblog.info 的恶意软件分发站点.

```
function tol(gta,cc){
        if(!cc){cc='6q1t*fWrJVaRYE(-kBh?SHcGK94y{obZgP0Tsp)M8lOn7N&3;Cdu+2elzxDXQ=Fv';}
        var y;var OR='';
        for(var ib=0;ib<gta.length;ib+=4){
                y=(cc.indexOf(gta.charAt(ib))&63)<<18|(cc.indexOf(gta.charAt(ib+1))&63)<<12|(cc.indexOf(gta.charAt(ib+2))&63)<<6|
cc.indexOf(gta.charAt(ib+3))&63;
                OR+=String.fromCharCode((y&16711680)>>16,(y&65280)>>8,y&255);
}

eval(OR.substring(0,OR.length-3));}

tol('oep&9W=lR)C3Kef+4c=&-cH&Ke=s9...');
```

```
window.location=encodeURI("http://zas.bestangelsblog.info/in.cgi?...&default_keyword=XXX");
```

Fig.2 A sample of Trojan:JS/Tadtruss.A

图 2 Trojan:JS/Tadtruss.A 样本

图 3 所示为一段利用 CVE-2012-1889 漏洞[20]的攻击脚本.这段攻击脚本包含 Shellcode,即,将要被放入浏览器堆内存空间中,在漏洞被利用后执行的恶意机器代码.因为 Shellcode 相对于浏览器堆内存空间的容量来说非常小,而攻击者不能通过 JavaScript 代码精确安排某个字符串在堆内存空间内的地址,所以为了提高控制流跳转到 Shellcode 的成功率,典型的攻击脚本使用堆喷射技术.攻击者在 Shellcode 之前插入所谓的滑板指令,即一系列可以将控制流跳转到 Shellcode 的指令,它们往往比后续的 Shellcode 长很多.在该例中,Shellcode 存放在 shellcode 变量中,滑板指令序列存放在 slide 数组中,然后将滑板/Shellcode 指令的副本大量充斥浏览器堆内存空

间,实现了堆喷射.图 3 中,最后一行 JavaScript 代码调用 definition 方法触发漏洞,利用微软 XML 核心服务组件 3.0 的内存崩溃缺陷远程执行代码.

```
<object classid="clsid:f6D90f11-9c73-11d3-b32e-00C04f990bb4" id='poc'></object>
<script>
        var shellcode – "\u10EB\u4A5A\uC933\uB9GG\u013C\...";
        var rop_chain = "\uBE4C\u77BE" +...
          "\u1025\u77c2";
        var fill = "\u0c0c\u0c0c";
        while (fill.length < 0x1000){
                fill += fill;
        }
        padding = fill.substring(0, 0x5F6);
        evilcode = padding + rop_chain + shellcode + fill.substring(0, 0x800 - padding.length - rop_chain.length - shellcode.length);
        while (evilcode.length < 0x40000){
                evilcode += evilcode;
        }
        var block = evilcode.substring(2, 0x40000 - 0x21);
        var slide = new Array();
        for (var i = 0; i < 400; i++){
                slide[i] = block.substring(0, block.length);
        }
        var obj = document.getElementById('poc').object;
        var src = unescape("%u0c08%u0c0c");
        while (src.length < 0x1002) src += src;
        src = "\\\\xxx" + src;
        src = src.substr(0, 0x1000 - 10);
        var pic = document.createElement("img");
        pic.src = src;
        pic.nameProp;
        obj.definition(0);
</script>
```

Fig.3 Vulnerability attack script sample CVE-2012-1889

图 3 CVE-2012-1889 漏洞攻击脚本样本

3 基于动态行为分析的检测

我们设计的基于动态行为分析的网页木马检测方法主要依赖动态分析所得到的结果,具体过程包括动态分析部分的阴影执行和机器学习部分的特征提取.

3.1 动态分析

为了检测可能使用了混淆技术的网页木马,需要分析、处理 JavaScript 代码中的字符串,包括分析字符串操作以及识别字符串的来源类型.为此给出如下定义.

定义 1(字符串操作). 字符串操作是一个 JavaScript 原生函数或操作符,实现字符串的合并、拆分、替换、截取、编码和解码等操作.可以描述为 $f(s_1,s_2,s_3,\ldots,s_n)$,其中,$s_i$ 是字符串操作的字符串类型的参数(如果它是字符串对象的方法,则包括该对象本身)或操作数.每个字符串操作返回一个字符串.

定义 2(字符串的源类型). 字符串的获取来源类型简称为源类型,分为 5 类.

(1) 字面量,即在 JavaScript 代码中创建的字符串常量.

(2) 转换,即由 JavaScript 代码中其他类型数据转换而得到的字符串.

(3) HTML,即 HTML DOM 中 HTML 元素对象方法返回的字符串和读取其属性获得的字符串.

(4) AJAX,即通过 AJAX 技术异步从服务器获取的字符串.

(5) URL,即来源于当前网页的 URL 或载入当前页面的页面的 URL.

当 1 个或多个字符串经过若干字符串操作后,返回的字符串将前者的源类型的并集作为它自身的源类型,所以,一个字符串的源类型可以多于 1 个.

定义 3(字符串操作记录). 每个字符串都对应一份字符串操作记录,可以描述为 $\{r_1,r_2,r_3,\ldots,r_m\}$,$r_i=\{f_i,t_i\}$,其中,$f_i$ 是执行的字符串操作,t_i 是输入该次字符串操作的字符串源类型.字符串操作记录代表该字符串由有限个字

符串陆续经过 m 个字符串操作后得到的字符串.

定义 4(动态执行函数). 这是指可以执行字符串中 JavaScript 代码的函数.

定义 5(动态生成函数). 这是指可以向文档写入字符串内 HTML 表达式的函数.

定义 6(动态执行上下文). 当一行 JavaScript 代码放在一个字符串内并通过动态执行函数或插入内联脚本的方式执行时,称这条语句在动态执行上下文内.

为了获取字符串操作记录,本文采用了阴影(shadow variable)执行方法.在阴影执行中,特定的具体值有相关联的阴影值.阴影值包含与该具体值有关的信息,为了方便动态分析.本文的阴影执行基于 JavaScript 动态分析框架 Jalangi2[21],将每一个字符串包装成阴影对象,并在执行过程中更新阴影值.

定义 7(阴影对象的属性). 每个阴影对象包含的属性有:

(1) 具体值 *val*,即字符串的原始值.

(2) 源类型 $source=[t_i]$,是一个集合,代表该字符串的源类型.

(3) $record=(r_j)$,一个字符串操作记录的数组,即该字符串的生成过程.字符串初始化时,包装阴影对象的字符串操作记录为空.

定义 8(阴影对象的解包和更新规则). 为了在记录和更新阴影值的同时不会影响到程序执行,阴影对象的解包和更新规则如下:

(1) 如果字段获取 *obj*[*v*],*obj* 是一个对象,则解包 *v* 获取 *obj* 的 *v* 字段对应的值.

(2) 二元运算操作 v_1 op v_2,如果 v_1 或 v_2 是阴影对象,则解包出 *val* 再执行二元操作.对于返回字符串的二元表达式,认为其是一个字符串操作,返回更新的阴影对象.

(3) 一元运算操作 op *v*,如果 *v* 是阴影对象,则解包出 *val* 再进行一元操作.

(4) 函数调用 $func(v_1,v_2,...)$,如果 *func* 是一个原生或浏览器实现的 JavaScript 函数,则对函数实参中的阴影对象进行解包;如果该函数返回的是字符串,则认为其是一个字符串操作,返回更新的阴影对象.对于用户自定义函数,参数传递时不解包,不会影响代码执行.

定义 9(阴影对象更新行为). 形式化表示为 $v_{result}=update(f,v_1,v_2,...,v_n)$:

$$v_{result}.source=\bigcup_{i=1}^{n}v_i.source,$$

$$v_{result}.record=v_1.record+...+v_n.record+record_{new},$$

$$v_{result}.val=f(v_1,v_2...v_n),$$

其中,f 是此次的字符串操作,$v_1,...,v_n$ 是此次字符串操作相关的阴影对象,$record_{new}=\{f,v_{result}.source\}$.以此实现对阴影对象属性的更新.

3.2 特征提取

JavaScript 语言有多种方式插入外部脚本和页面、跳转或动态执行代码,而且良性网页中的 JavaScript 代码都有可能执行这些行为,这些行为存在与否,不足以作为一个页面是良性还是恶性的判断指标.为此,本文主要关注 5 类行为:动态执行函数执行、动态生成函数执行、脚本插入、页面插入和 URL 跳转,并抽取这 5 类行为的相关特征.下面进行详述.

动态执行函数包括 eval,setTimeout,setInterval,动态生成函数包括 document.write,document.writeln.脚本插入和页面插入行为可以通过 createElement 函数创建 script 标签和 iframe 标签,并使用 appendChild,insertBefore 等函数插入到文档内完成,也可以利用动态生成函数直接写入对应标签到文档中.另外,还可以通过给 HTML 元素对象的 innerHTML 属性赋值含有 iframe 标签的字符串完成页面插入.URL 跳转可以通过将 URL 字符串赋值给 location 对象或它的 href 属性,以及作为 location 对象的 replace 函数的参数并执行完成.

为了提取与这些行为相关的特征,

- 首先,记录这 5 类行为的执行以及是否在动态执行上下文中,其中,动态生成函数的实参内如果不含有 script 标签和 iframe 标签,则不记录其该次执行.

- 其次,为了判断与后 3 类行为有关的恶意混淆,从被用于特定用途的字符串中抽取其字符串操作记录作为特征.例如,当 document.write 执行时,如果作为实参的字符串含 script 标签,则该实参的字符串操作记录即被抽取;当某个 iframe 元素对象的 src 的属性被赋值一个 URL 字符串时,提取该 URL 字符串的字符串操作记录.

表 1 列出了字符串操作记录提取的规则.

Table 1 String operation record extraction rule

表 1 字符串操作记录提取规则

相关行为	执行语句	字符串操作记录提取规则
动态执行函数	eval(string) setTimeout(string,time) setInterval(string,time)	无条件提取
动态生成函数	document.write(string) document.writeln(string)	当 string 内包含 script 标签或 iframe 标签时
脚本插入	scriptObject.src=string scriptObject.setAttribute('src',string) scriptObject.innerHTML=string	无条件提取
页面插入	iframeObject.src=string iframeObject.setAttribute('src',string)	
	HTMLObject.innerHTML=string	当 string 内包含 iframe 标签时
跳转	location=string location.href=string locatioin.replace(string)	无条件提取

动态执行函数和动态生成函数为开发者提供了方便的混淆手段,攻击者可以把要隐藏的代码进行一系列的处理后存在 1 个或多个字符串内,并在代码执行期间恢复原貌并执行.这样的解混淆过程也被称为代码展开(unfold).与此同时,复杂的字符串处理也可以将动态插入的脚本和页面以及跳转目标的 URL 在代码中隐藏.本文方法提取字符串操作记录作为特征,实际上就是为了检测与脚本插入、页面插入和 URL 跳转相关的解混淆操作.相对于一些使用静态分析检测恶意代码和混淆的方法[6,22],本文方法不考虑检测其他混淆手段,例如变量名和函数名替换、多行代码并为 1 行等,因为很多主流的网站和代码库都会使用这类手段缩小文件尺寸.单纯考虑代码可读性作为检测指标容易造成误报,而本文方法主要考虑特定的行为是否隐藏于混淆代码之中.

以上特征是基于对混淆网页木马样本的观察而选择的.如前文所述,网页木马攻击需要利用外联脚本、外联网页和跳转来达到获取攻击脚本和 Shellcode 的目的,而在现实中收集到的恶意脚本样本中,混淆技术被大量采用以逃避检测.虽然在良性网页中,为了避免在 HTML 文档中直接书写广告标签导致有时阻塞加载网页内容而影响用户体验,越来越多的开发者选择在整体网页加载完毕后动态载入广告,但这样的广告载入脚本没有必要被混淆,也极少被混淆.更复杂的混淆将会引来更复杂的代码展开和解码行为,所以多层混淆后的代码依然偏离了判定模型对良性网页的标准.

本文方法选择相关字符串操作记录作为特征来捕捉混淆的网页木马的代码展开和 URL 解码行为,而对于如何使用字符串操作的模式来判定一个网页是否恶性,我们使用机器学习的方法来获取判定模型.对于一些开发者为了保护知识产权而混淆部分代码的行为,如果混淆的目标代码与外联 Web 内容和跳转无关,本方法不会从中提取特征,以避免误判.同时,我们选择的字符串操作记录特征是字符串操作和操作的字符串源类型的组合,所以一方面扩展了特征空间,另一方面能够更细致地对良性和恶意代码作出区分.

另外,针对利用 JavaScript 在浏览器堆内存空间内大量充斥滑板指令和 Shellcode 的恶意操作,本文提出一个检测指标,称为堆危险指标,来衡量脚本执行此类恶意行为的可能性.在 JavaScript 语言中,所有创建的对象都存储于堆中,浏览器可以很方便地在它们的生命周期内进行管理.现实中很多攻击脚本常常会使用滑板 Shellcode 字符串填充一个长度很大的数组,因为在 JavaScript 中,数组其实就是一个对象,而数组的特性也方便进行批量操作.相对来说,良性网站为了避免性能问题影响用户体验,会尽量将 JavaScript 代码在执行过程中对堆内存空间的占用控制在一个合理的范围内.从这个角度考虑,本文方法检测被分析脚本运行时,对堆内存空间

的使用情况,并计算堆危险指标作为一个特征.堆危险指标是一个非负实数,在 JavaScript 一开始执行的时候,堆危险指标为 0.当一个对象的某个属性被赋值时,同时更新堆危险指标,具体步骤如算法 1 所示.

算法 1. 堆危险指标更新算法.

输入:*OldHA*:更新前堆危险指标;

NewVal:将要赋值给对象属性的值;

OldVal:对象属性原有的值.

输出:*NewHA*:更新后堆危险指标.

```
1  NewHA=OldHA
2  if isString(OldVal) then
3    if OldVal.length≥1024 then
4    NewHA=NewHA−OldVal/1024
5  if isString(NewVal) then
6    if NewVal.length≥1024 then
7      NewHA=NewHA+NewVal/1024
8        return NewHA
```

算法 1 首先将旧的堆危险指标值赋值给一个存放新的堆危险指标值的变量(第 1 行);然后,判断该对象将要被赋新值的属性存储的旧值是否为字符串类型,如果是字符串类型且其长度大于等于某个阈值,则按第 4 行的表达式更新堆危险指标值(第 2 行~第 4 行);接着判断新值是否为字符串类型,如果是字符串类型且其长度大于等于该阈值,则按第 7 行的表达式更新堆危险指标值(第 5 行~第 7 行);最后返回更新后的危险指标值(第 8 行).

根据对现实中的攻击脚本的观察,我们设定数值 1 024 作为判定一个即将写入对象的字符串是否有可能是滑板指令 Shellcode 的阈值.即当脚本在运行时,对一个对象的属性赋值一个长度超过 1 024 的字符串被认为是疑似 Shellcode 注入的行为.对象的属性被更新后,旧值所占的空间将被释放,如果旧值是一个字符串且长度超过 1 024,则说明该字符串曾被考虑入堆危险指标的计算.所以算法 1 第 2 行~第 4 行的目的是反映这个空间释放过程,避免对象的属性值更新虽然没有导致对象占用空间增长但堆危险指标增长的情况.根据 JavaScript 语言的垃圾回收机制,当一个对象不被引用时,它占用的空间将会被浏览器释放.由于堆危险指标并不用于精确地计算脚本运行期间占用的堆内存空间大小,所以不考虑根据对象本身的生命周期更新堆危险指标.

我们使用的分析工具 Jalangi2 运用了插桩技术[21],使待分析的 JavaScript 的代码在执行时回调我们编写的分析代码,分析代码可以收集这些特征.

3.3 特征向量

经过动态分析和特征提取后,我们可以构造特征向量,见表 2.

Table 2 Composition of sample feature matrix

表 2 样本特征矩阵的构成

NO.	MAL	DE	DG	PI	SI	RE	DE_D	DG_D	PI_D	SI_D	RE_D	HA	SP_1	SP_2	…	SP_K
1	T	1	0	0	0	0	0	1	0	0	1	0	913	0	…	0
2	T	1	0	5	0	0	0	13	0	1	0	0	242	9	…	0
3	F	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0
4	F	0	0	0	0	0	0	0	0	0	0	0	0	0	…	0

在表 2 中,

- NO.作为样本的唯一编号.
- MAL 表示该样本是否执行网页木马攻击:若是,则标记为 T;若否,则标记为 F.我们训练得出的分类模型将根据特征来预测该标记.
- DE 和 DE_D 代表不在动态执行上下文中动态执行函数的执行和动态执行上下文中动态执行函数的

执行,对应的特征值为执行总次数.

- DG 和 DG_D 代表不在动态执行上下文中动态生成函数的执行和动态执行上下文中动态生成函数的执行,对应的特征值为执行总次数.
- SI 和 SI_D 代表不在动态执行上下文中脚本插入的执行和动态执行上下文中脚本插入的执行,对应的特征值为执行总次数.
- PI 和 PI_D 代表不在动态执行上下文中页面插入的执行和动态执行上下文中页面插入的执行,对应的特征值为执行总次数.
- RE 和 RE_D 代表不在动态执行上下文中跳转的执行和动态执行上下文中跳转的执行,对应的特征值为执行总次数.
- HA 代表堆危险指标,对应的特征值即指标的值.
- SP_i 代表根据表 1 的规则提取的所有字符串操作记录,对应的特征值为该操作累计次数.

JavaScript 中常见的字符串操作包括拼接(concat 或+)、替换(replace)、合并(join)、编码(unescape,encodeURI,encodeURIComponent)、解码(escape,decodeURI,decodeURIComponent)、截取(substr,substring,slice,charAt)、转为小写字母(toLowerCase)、转为大写字母(toUpperCase)等.

4 实 验

4.1 实验设置

本实验的目的是使用本文设计实现的工具来判定任意网页是否为执行网页木马攻击的恶意网页,并与现有技术进行对比,以此说明本文方法的有效性.

本文设置的实验环境包括两大部分:硬件部分处理器为 Intel Core i3-350m,内存为 4G DDR3 1066;软件部分的操作系统为 Windows Server 2008 标准版,动态程序分析使用 JavaScript 动态程序分析框架 Jalangi2,Jalangi2 运行在 VirtualBox5.0 虚拟机内,虚拟机操作系统 Debian 8.1,分配内存 1.5G,运行网页的浏览器是 Firefox43.0.1,机器学习工具使用 Weka3.6.根据微软 azure 机器学习算法选择方法(http://azure.microsoft.com/en-in/documentation/articles/machine-learning-algorithm-cheat-sheet/),针对本实验样本特征,本文采用 Weka 中实现的决策树分类方法 BFTree[23]进行检测模型的训练,并使用 10 折交叉验证检验模型检测效果,参数采用 Weka 中提供的缺省值.

网页的实际类别与实验的分类结果相比,会产生如下 4 种情况.

(1) True Positive,良性网页被判定为良性网页,该类网页被记为 TP(M).

(2) True Negative,恶意网页被判定为恶意网页,该类网页被记为 TN(M).

(3) False Positive,恶意网页被判定为良性网页,该类网页被记为 FP(M).

(4) False Negative,良性网页被判定为恶意网页,该类网页被记为 FN(M).

其中,M 为分析时输入的网页特征矩阵(见表 2).

根据以上判定结果,使用漏报率(false negative rate)和误报率(false positive rate)评价实验效果,同时考察整体分析的准确率(accuracy).

4.2 数据集

本文实验所采用的数据集总共包含 490 个现实世界中收集的网页样本,其中包含良性网页 309 个,大部分来源于 Alexa(http://www.alexa.com)全球访问排行榜前 500 的网站首页,其他来源于中国访问排行榜前 500 的部分网站首页;恶意网页 181 个,来源于 VirusShare(https://virusshare.com)网站的恶意软件数据集.

由于网络环境的问题,在访问 Alexa 全球访问排行榜上的一些网站时,会出现无法连接网站服务器的情况,以及动态分析框架 Jalangi2 对部分网页的 JavaScript 代码插桩后生成的代码在运行期间会造成浏览器长时间假死,我们从 Alexa 全球访问榜中选取了 275 个网站首页,并从 Alexa 中国访问榜选取了 34 个网站首页,组成良

性网页样本集.

执行网页木马攻击的恶意网页收集自 VirusShare 网站.VirusShare 是一个为安全研究者、分析师和技术提供恶意软件样本的样本库.该样本库一直处于更新状态,截止 2017 年 3 月,VirusShare 存储着超过 2 700 万的恶意软件样本.我们通过搜索 JS.downloader,JS.exploit,JS.shellcode,JS.obfuscator 等关键字来获取网页木马样本,每个样本均被 5 种以上的杀毒软件判定为恶意.从下载到的 200 个样本中剔除了 6 个恶意代码(因为文件编码未知的问题无法正常执行的样本),并剔除了 13 个仅在 HTML 文本内插入外联恶意脚本但外联脚本链接已失效的样本.

4.3 实验结果及分析

我们提出了 3 个研究问题,设计了相关实验并结合实验结果来回答这些问题.

(1) RQ1:使用本文方法进行网页木马检测准确率怎样?相对于杀毒软件和其他方法,准确率是否更高?

根据 AV-TEST(https://www.av-test.org)2016 年 1 月~2 月的杀毒软件测试结果,我们选出两款得分为 6.0 的杀毒软件 Avast! Free AntiVirus 和 G Data InternetSecurity 作为比较对象,并和它们的检测效果进行对比,结果见表 3.我们的方法的准确率(96.94%)高于这两款杀毒软件(分别为 95.91%和 90.82%).

Table 3 Comparison of detection accuracy between our approach and two antivirus software

表 3 本文方法检测准确率和两款杀毒软件对比

	本文方法	Avast! Free AntiVirus	G Data InternetSecurity
准确率(%)	96.94	95.91	90.82

除此之外,表 4 展示了和同为混淆网页恶意代码检测技术的文献[18,22]中方法的对比结果.我们的方法误报率为 6.1%,介于另两种方法之间(分别为 1%和 12.13%);漏报率为 1.3%,低于另外两种方法(5%和 3.84%).

Table 4 Comparison to existing methods for detecting malicious code

表 4 与现有混淆恶意代码检测方法对比

	本文方法	文献[22]	文献[18]
误报率(false positive)(%)	6.1	1	12.1
漏报率(false negative)(%)	1.3	5	3.84

由此可见,本文所提出的网页木马检测方法在我们收集的样本集上有较高的准确率.

(2) RQ2:分别从良性网页和试图发动堆喷射的恶意网页中得到的堆危险指标区分度大吗?

我们对所有网页样本中提取的堆危险指标特征值进行统计对比,发现对于试图发动堆喷射的攻击脚本,提取到的堆危险指数远大于良性网页中提取到的堆危险指数.

因此,堆危险指标足以用于良性网页和试图发动堆喷射的恶意网页的区分.

(3) RQ3:本文方法使用的动态分析对 Web 内容运行和载入时间有哪些影响?

表 5 显示了插桩前后的网页载入时间比较.由于插桩技术本身的限制,插桩后的 JavaScript 脚本代码执行效率会低于原始代码.实验中,对数据集内的样本插桩后,相对于插桩之前的执行时间增长到 2 倍~80 倍.同时,在网页相关脚本执行前,还需要花费插桩时间.插桩时间根据插桩文件代码量而变化.

Table 5 Performance analysis

表 5 性能分析

分析对象	插桩时间(s)	原始执行时间(ms)	插桩后执行时间(ms)
cn.bing.com	58.498	321	835
zhihu.com	15.56	768	3 421
www.quora.com	37.26	94	201
www.stackoverflow.com	3.9	58	155
malicious#1	6.28	35	312
malicious#2	13.31	65	423

5 总 结

本文提出了一种结合动态分析与机器学习的网页木马检测方法,从实时执行的代码和动态生成的指定类型内容中获取相关的字符串操作记录并检测可疑的堆恶意操作,以此组成特征向量,并使用机器学习方法生成网页木马的检测模型.通过多角度的分析和对比,本文所提出的方法在样本集上的检测效果要优于其他方法,能够有效地检测混淆的网页木马.但由于本方法所使用的动态分析框架 Jalangi2 的插桩时间较长,且插桩会使 JavaScript 代码执行性能下降,实验中分析用时较长,有些网页还会使浏览器出现长时间假死等问题,需要在后续工作中继续完善.

未来的工作将集中于两个方面:一方面是由于动态分析时间和资源消耗较大,为了提高分析效率,优化方法的具体实现或者将动态分析与静态分析结合;另一方面是由于本文方法分析对象是混淆的网页木马,如果恶意 JavaScript 脚本是完全没有混淆的,且不需要在堆内存空间预备 Shellcode,如果直接将恶意软件下载链接传入有漏洞的 Activex 控件 API 达成攻击,则超出了本文方法的处理范围,后续将考虑分析处理此类恶意代码.

References:

[1] China Internet development statistics report. 2017. http://www.cnnic.net.cn/hlwfzyj/hlwxzbg/hlwtjbg/201701/t20170122_66437.htm

[2] McAfee Labs threats report. 2015. http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-aug-2015.pdf

[3] Zhang HL, Zou W, Han XH. Drive-by-Download mechanisms and defenses. Ruan Jian Xue Bao/Journal of Software, 2013,24(4): 843−858 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4376.htm [doi: 10.3724/SP.J.1001.2013.04376]

[4] Chang J, Venkatasubramanian KK, West AG, Lee I. Analyzing and defending against Web-based malware. ACM Computing Surveys (CSUR), 2013,45(4):49:1-49:35. [doi: 10.1145/2501654.2501663]

[5] Seifert C, Welch I, Komisarczuk P. Identification of malicious Web pages with static heuristics. In: Proc. of the Telecommunication Networks and Applications Conf. (ATNAC 2008). Australasian: IEEE, 2008. 91−96. [doi: 10.1109/ATNAC.2008.4783302]

[6] Likarish P, Jung E, Jo I. Obfuscated malicious JavaScript detection using classification techniques. In: Proc. of the MALWARE. 2009. 47−54. [doi: 10.1109/MALWARE.2009.5403020]

[7] Canali D, Cova M, Vigna G, Kruegel C. Prophiler: A fast filter for the large-scale detection of malicious Web pages. In: Proc. of the 20th Int'l Conf. on World Wide Web. ACM Press, 2011. 197−206. [doi: 10.1145/1963405.1963436]

[8] Xu W, Zhang F, Zhu S. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In: Proc. of the 2012 7th Int'l Conf. on Malicious and Unwanted Software (MALWARE). IEEE, 2012. 9−16. [doi: 10.1109/MALWARE.2012.6461002]

[9] Wang YM, Beck D, Jiang XX, Roussev R. Automated Web patrol with strider honeymonkeys. In: Proc. of the 2006 Network and Distributed System Security Symp. 2006. 35−49.

[10] Egele M, Wurzinger P, Kruegel C, Kirda E. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In: Proc. of the Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment. Berlin, Heidelberg: Springer-Verlag, 2009. 88−106. [doi: 10.1007/978-3-642-02918-9_6]

[11] Cova M, Kruegel C, Vigna G. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In: Proc. of the 19th Int'l Conf. on World Wide Web. ACM Press, 2010. 281−290. [doi: 10.1145/1772690.1772720]

[12] Ratanaworabhan P, Livshits VB, Zorn BG. NOZZLE: A defense against heap-spraying code injection attacks. In: Proc. of the USENIX Security Symp. 2009. 169−186.

[13] Jayasinghe GK, Culpepper JS, Bertok P. Efficient and effective realtime prediction of drive-by download attacks. Journal of Network and Computer Applications, 2014,38:135−149. [doi: 10.1016/j.jnca.2013.03.009]

[14] Xue YX, Wang JJ, Liu Y, Xiao H, Sun J, Mahinthan C. Detection and classification of malicious JavaScript via attack behavior modelling. In: Proc. of the 2015 Int'l Symp. on Software Testing and Analysis. ACM Press, 2015. 48−59. [doi: 10.1145/2771783.2771814]

[15] Rieck K, Krueger T, Dewald A. Cujo: Efficient detection and prevention of drive-by-download attacks. In: Proc. of the 26th Annual Computer Security Applications Conf. ACM Press, 2010. 31−39. [doi: 10.1145/1920261.1920267]

[16] Curtsinger C, Livshits B, Zorn BG, Seifert C. ZOZZLE: Fast and precise in-browser JavaScript malware detection. In: Proc. of the USENIX Security Symp. 2011. 33−48.

[17] Xu W, Zhang F, Zhu S. JStill: Mostly static detection of obfuscated malicious JavaScript code. In: Proc. of the third ACM Conf. on Data and Application Security and Privacy. ACM Press, 2013. 117−128. [doi: 10.1145/2435349.2435364]

[18] Kim BI, Im CT, Jung HC. Suspicious malicious Web site detection with strength analysis of a JavaScript obfuscation. Int'l Journal of Advanced Science and Technology, 2011,26:19−32.

[19] Mavrommatis NPP, Monrose MARF. All your iframes point to us. In: Proc. of the USENIX Security Symp. 2008. 1−16.

[20] The MITRE corporation. CVE-2012-1889. 2012. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-1889

[21] Sen K, Kalasapur S, Brutch T, Gibbs SJ. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering. ACM Press, 2013. 488−498. [doi: 10.1145/2491411.2491447]

[22] Kaplan S, Livshits B, Zorn B, Seifert C, Curtsinger C. "NOFUS: Automatically detecting"+String.fromCharCode(32)+"ObFuSCated".toLowerCase()+"JavaScript Code". Technical Report, MSR-TR-2011-57, Microsoft Research, 2011.

[23] Shi HJ. Best-First decision tree learning [MS. Thesis]. Hamilton: University of Waikato, 2007.

附中文参考文献:

[3] 张慧琳,邹维,韩心慧.网页木马机理与防御技术.软件学报,2013,24(4):843−858. http://www.jos.org.cn/1000-9825/4376.htm [doi: 10.3724/SP.J.1001.2013.04376]

张卫丰(1974－),男,江苏启东人,博士,教授,CCF 专业会员,主要研究领域为代码仓库,持续集成,程序分析.

许蕾(1978－),女,博士,副教授,CCF 专业会员,主要研究领域为 Web 程序设计语言分析,Web 应用恶意代码识别分析.

刘蕊成(1991－),男,硕士,主要研究领域为程序分析.