CHAPTER 2

Security of Browser

In recent years, with the development of the Internet, it can be said that the browser is the biggest entrance to the Internet; the vast majority of users access the Internet using the browser. This has resulted in a tremendous rise in the browser market.

In this highly competitive environment, more and more people have taken security of the browser seriously. On one hand, the browser is inherently a client; it will be quite safe if equipped with safety features like security software. On the other hand, security of browser has become a competing factor for browser vendors, who hope to establish technical barriers for security to gain a competitive advantage.

Therefore, in recent years with constantly updated browser versions, browser security features are becoming more powerful. In this chapter, we will introduce some major browsers' security features.

2.1 SAME-ORIGIN POLICY

The same-origin policy is a core convention of browsers; it is also the most basic security function. If the same-origin policy is not available, the browser's normal function may be affected. The web is built on the basis of the same-origin policy, but a browser is just an implementation strategy for the same-origin policy.

For client-side web security, in-depth understanding of the same-origin policy is very important to handle unforeseen problems. Mostly, the same-origin policy implementation is recessive and transparent. Many of the issues from the same-origin policy are not easy to present the problem; if you are not familiar with the same-origin policy, you may always not understand the problem and the reason.

Browsers' same-origin policy limits *document* from different sources or scripts, and does allow reading or setting certain properties for the current *document*.

This strategy is extremely important. Imagine this: If there is no same-origin policy, the section of JavaScript at a.com, when b.com is not loading this script, can alter the b.com page (in the browser's display). In order to avoid such chaotic behavior of the browser page, the browser presents the concept of *origin* (source) so that objects from different origins cannot interfere with one another.

TABLE 2.1 The Examples Show Up Same-Origins or Different Origins.

URL	Outcome	Reason
http://store.company.com/dir2/other.html	Success	
http://store.company.com/dir/inner.another.html	Success	
http://store.company.com/secure.html	Failure	Different protocol
http://store.company.com:81/dir./etc/html	Failure	Different port
http://store.company.com/dir/other.html	Failure	Different host

JavaScript examples with the same-origin policy are listed in Table 2.1.

Table 2.1 shows that the factors that have an effect on the *source* are host (domain name or IP address, if the IP address is seen as a root domain), subdomain, port, and protocol.

It should be noticed that, for the current page, the domain that stores the page JavaScript file is not important; the domain loading the JavaScript page matters much.

In other words, using the following code, `a.com` loaded `b.js` on `b.com`:

```
<script src = http://b.com/b.js ></script>
```

but `b.js` is running at a.com, so for the current page (a.com page), the origin of b.js should be `a.com` rather than b.com.

In the browser, `<script>`, `<img>`, `<iframe>`, `<link>`, and many other labels can be loaded through cross-domain resources without restrictions from the same-origin policy. When every time attributes with an "src" label are loaded, the browser actually initiates a GET request. Unlike XMLHttpRequest, for the resource loaded via the src attribute resource, the browser limits the authority of JavaScript so that it cannot read or write returns.

For XMLHttpRequest, it can get access to the contents of the object from the same origin. For example:

```
<html>
<head>
<script type="text/javascript">
var xmlhttp;
function loadXMLDoc(url)
{
xmlhttp=null;
if (window.XMLHttpRequest)
  {// code for Firefox, Opera, IE7, etc.
  xmlhttp=new XMLHttpRequest();
  }
else if (window.ActiveXObject)
  {// code for IE6, IE5
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
  }
```

```
if (xmlhttp!=null)
  {
  xmlhttp.onreadystatechange=state_Change;
  xmlhttp.open("GET",url,true);
  xmlhttp.send(null);
  }
else
  {
  alert("Your browser does not support XMLHTTP.");
  }
}
function state_Change()
{
if (xmlhttp.readyState==4)
  {// 4 = "loaded"
  if (xmlhttp.status==200)
    {// 200 = "OK"
    document.getElementById('T1').innerHTML=xmlhttp.
      responseText;
    }
  else
    {
    alert("Problem retrieving data:" + xmlhttp.statusText);
    }
  }
}
</script>
</head>
<body onload="loadXMLDoc('/example/xdom/test_xmlhttp.txt')">
<div id="T1" style="border:1px solid black;height:40;width:300;padd
  ing:5"></div><br />
<button onclick="loadXMLDoc('/example/xdom/test_xmlhttp2.
  txt')">Click</button>
</body>
</html>
```

But XMLHttpRequest is limited by the same-origin policy and cannot get access to a cross-domain resource, especially in AJAX application development.

However, the Internet is open; as your business grows, demand for cross-domain requests increases. For this purpose, the W3C Committee developed a standard XMLHttpRequest cross-domain access. It will decide whether to allow cross-domain access through HTTP headers returned by the target domain, because for JavaScript, the HTTP header generally cannot be controlled. It is worth noting that the security foundation of this cross-domain access is based on the trust that "JavaScript cannot control the HTTP header"; if this does not hold, the program will no longer be safe (Figure 2.1).

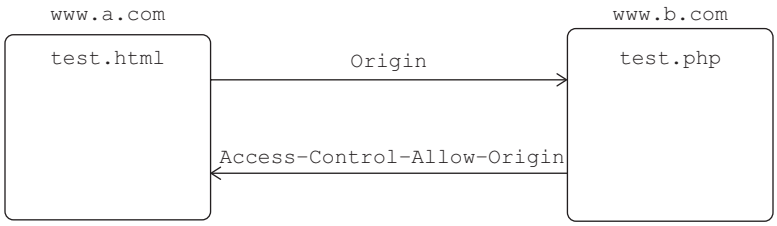

FIGURE 2.1 Cross-domain access request process.

For more information about the implementation process, please refer to Chapter 6.

A browser's document object model (DOM), Cookie, and XMLHttpRequest will be subject to restrictions by the same-origin policy, but third-party browser plug-ins may also have their own same-origin policies. Some of the most common plug-ins such as Flash, Java Applet, Silverlight, and Google Gears also have their own control strategy.

Take Flash, for example; it determines whether to allow the current *source* of Flash cross-domain access to target resources mainly through the crossdomain.xml file provided by the target site.

Take www.qq.com policy file, for example when the browser loads the Flash page in any other domain and access to www.qq.com is issued, Flash will first check if this policy file exists on www.qq.com. If yes, Flash will check whether the requesting domain is in the permitted range (Figure 2.2).

In this strategy document, only the requests from the domains "*. qq.com" and "*. gtimg.com" are allowed. In this way, the security in Flash can be managed at the origin

In Flash 9 and later versions, a multipurpose Internet mail extensions (MIME) check is used to make sure crossdomain.xml is legitimate, such as checking whether the content type which the server returns to the HTTP header is text/*, application/xml, or application/xhtml + xml. The reason why this should be done is that the attacker can control the behavior of Flash from uploading the crossdomain.xml file, bypassing the same-origin policy. Besides MIME checks, Flash also checks whether the crossdomain.xml is in the root directory, which can also lead to failure of some file inclusion attacks.

However, a browser with a same-origin policy is not always invincible, due to the realization of some of the problems. Some browsers with the same-origin policy have also been bypassed often, such as the cross-domain vulnerability in IE8 shown in Figure 2.2.

FIGURE 2.2 The crossdomain.xml file of www.qq.com.

www.a.com/test.html:

```
<body>
{}body{font-family:
aaaaaaaaaaaaa
bbbbbbbbbbbbbbb
</body>
```

www.b.com/test2.html:

```
<style>
@import url("http://www.a.com/test.html");
</style>
<script>
  setTimeout(function(){
    var t = document.body.currentStyle.fontFamily;
    alert(t);
  },2000);
</script>
```

In www.b.com/test2.html, CSS files such as http://www.a.com/test.html are loaded, rendering the current page into the DOM, and at the same time getting access to this content through document.body.currentStyle.fontFamily. If the problem occurs in IE's CSS parse process, IE will take the content behind fontFamily as a value and can read the content of www.a.com/test.html (Figure 2.3).

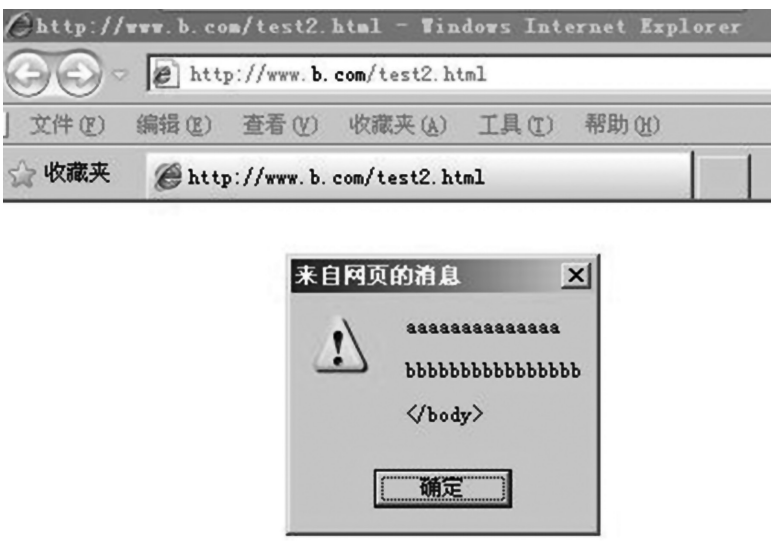

FIGURE 2.3 www.b.com can read the page content at www.a.com.

As mentioned before, tags like `<script>` can only load resources, not read or write the contents of the resource; however, this vulnerability could read the page content across domains. Therefore, it can bypass the same-origin policy and become a cross-domain vulnerability.

The same-origin policy is the basic security strategy of a browser. Many client-side scripting attacks must take this into account, which will be discussed in the following chapters. Once vulnerabilities in the same-origin policy occur and the policy is bypassed, it will bring serious consequences—all security solutions based on that same-origin policy will be compromised.

2.2 SANDBOX BROWSER

Client side attacks have increased a great deal in recent years (Figure 2.4).

Inserting some malicious code through browser vulnerabilities to execute arbitrary code attack is called *website embedded Trojan*.

Website embedded Trojan is a major threat that browsers face nowadays. Apart from antivirus software, browser vendors developed a number of techniques to counter website embedded Trojan.

For example, in Windows systems, browsers can defend memory attacks by closely combining the protection measures provided by the operating systems like data execution prevention (DEP), address space layout randomization (ASLR)., SafeSEH, etc. At the same time, browsers have also developed a multiprocess architecture, which greatly improved the security level.

Multiprocess architecture of a browser will separate each module and each browser instance; in this way, when a process crashes, it will not affect other processes.

Google Chrome is the first browser to adopt a multiprocess architecture. The main process of Google Chrome is divided into four: the browser process, the rendering process, the plug-in process, and the expansion process. Plug-in processes such as Flash, Java, PDF, etc., are distinctively isolated from the browser process and will not affect each other (Figure 2.5).

The rendering engine is isolated from the Sandbox. The web page code needs to communicate with the browser kernel process and the operating system only through the IPC channel, which will go through a number of security checks.

Sandbox, with the development of computer technology, is now generally referred to as *resource isolation class module*. Sandbox is designed to allow untrusted code to run in a certain environment, restricting it to access resources outside the quarantine area. If you

FIGURE 2.4 Websites attacked by website embedded Trojan on 2010.1~2013.6.

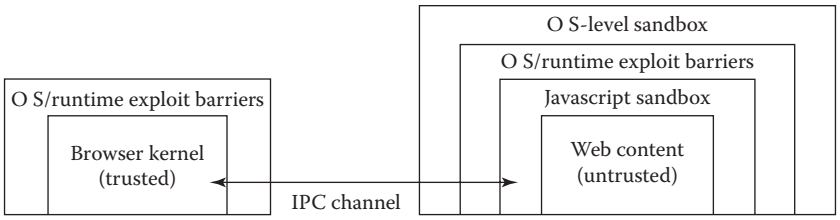

FIGURE 2.5 Google Chrome architecture.

must cross the border of Sandbox to generate data exchange, then data can only go through designated channels, for example, through encapsulated API in which the legality of the request will be strictly checked.

Sandbox is used in a wide range of applications. Take a shared hosting environment providing hosting services as an example: In order to prevent the user code from damaging the system environment or prevent the code from different users from affecting each other, a Sandbox should be used for isolating user codes in PHP, Python, Java, and the like. Sandbox needs to consider possible requests from user code in terms of the local file system, memory, databases, and networks. To achieve this, you can use the default deny policy or encapsulate the API.

With the use of the Sandbox technology, untrusted web page code and JavaScript code can run in a restricted environment to ensure the security of the local system.

A relatively complete Sandbox from Google Chrome is shown in Figure 2.6.

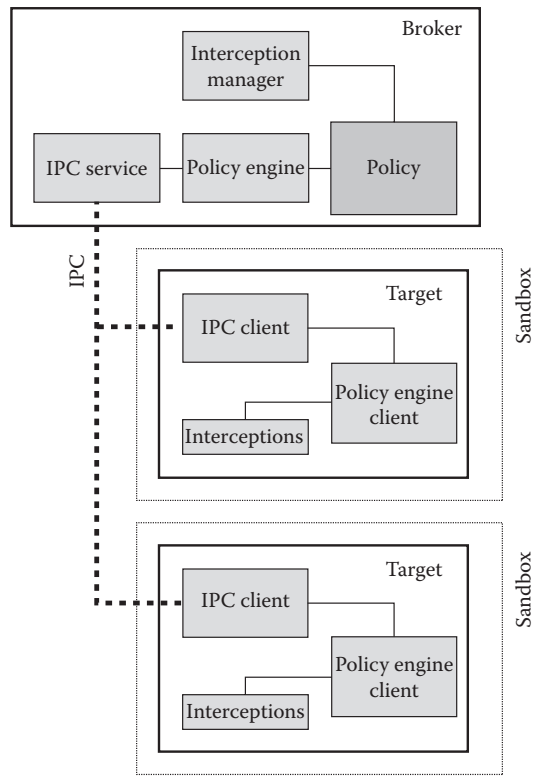

FIGURE 2.6 Google Chrome's Sandbox architecture.

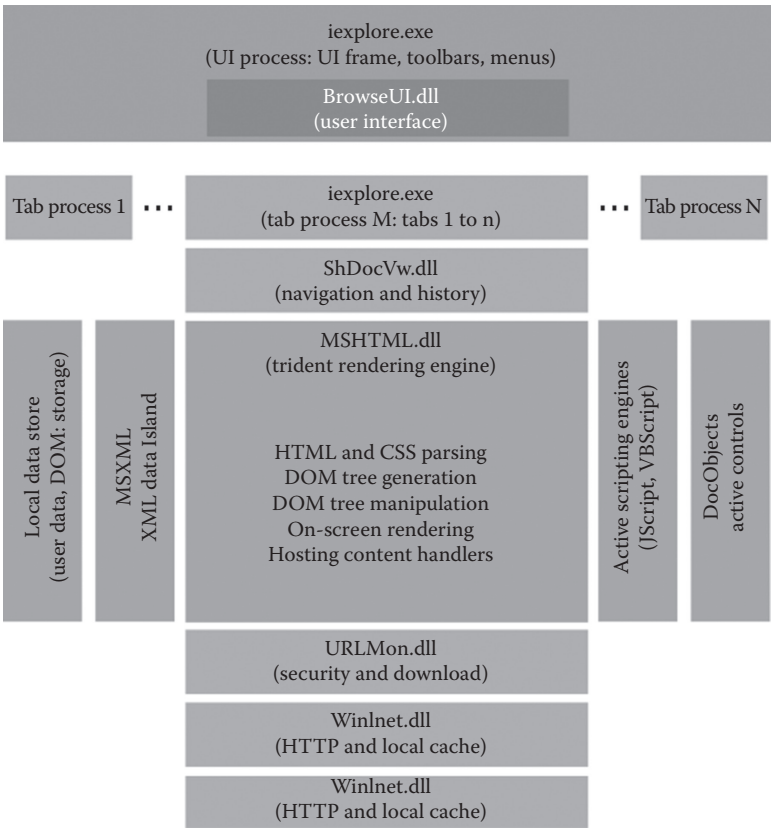

FIGURE 2.7 Architecture of IE8.

IE8 is a multiprocess architecture, in which each tab page is a separate process. IE8 architecture is shown in Figure 2.7.

Though the browsers today have multiple process architectures and Sandbox to ensure security, third-party plug-ins loaded by the browser can often bypass the Sandbox. For example, the browsers in the Pwn2Own conference were attacked due to loading of third-party plug-ins. Attacks using Flash, Java, PDF, and .Net Framework have become the trend in recent years.

Perhaps future browser security models will pay more attention to these third-party plug-ins. Browser vendors should work together to improve the standard of security strengthen their browsers.

2.3 MALICIOUS URL INTERCEPT

As mentioned in Section 2.2, *website embedded Trojan* attacks can destroy browser security; in many cases, when a website embedded Trojan attack is implemented, it will load a malicious website via `<script>`, `<iframe>`, etc., in a normal web page. Besides website embedded Trojan, there are various phishing and scam sites that could be dangerous to users. In order to safeguard users from such websites, browser manufacturers have launched applications to stop execution of malicious URLs, but again most of these security measures depend on the *blacklist*.

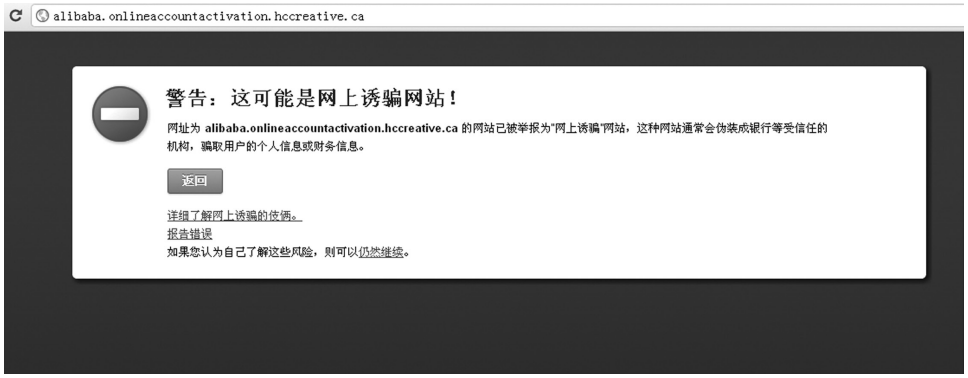

FIGURE 2.8 Warning from Google Chrome malicious URL.

Stopping malicious websites from opening can be simple. Usually, the browser periodically obtains an updated blacklist of malicious URLs from the server; if the users try to access a URL on this blacklist, the browser will return a warning page (Figure 2.8).

Malicious URLs can be divided into two categories: One category is sites embedded with Trojan—such sites often run malicious scripts, such as JavaScript or Flash, (including plug-ins and vulnerability from controls) containing shell code to implant a Trojan in the user's computer; the other is phishing sites—these sites imitate well-known, legitimate websites to trick users.

To identify these two kinds of sites, we need to establish many page characteristics based models, but these models are obviously not suitable to put on the client side, because it will enable the attackers to analyze, research, and bypass the rules. In addition, as browsers always have a huge user base, collecting users' visiting history also is an infringement of privacy, and the data quantity is too huge.

Because of these two reasons, browser vendors now mainly push the blacklist of malicious urls, which the browser blocks. It's rear to retrieve data from browser or build models at the user's side. Nowadays browser vendors work more with professional security vendors and use blacklist from these vendors or organizations.

Major browser vendors, such as Google and Microsoft, with strong R&D have lots of user data; they have their own security teams to conduct malicious website identification to obtain a blacklist. Blacklists are one of the core competencies for search engines as well.

PhishTank is an organization that provides free malicious URL blacklist, which receives contributions and updates from volunteers around the world (Figure 2.9).

Similarly, Google has also publicized its internal SafeBrowsing API, and any organization or individual can obtain the malicious URL blacklist. Apart from blocking websites on the blacklist, major browsers are beginning to support the EV SSL Certificate (extended validation SSL certificate) to enhance the identification of safe websites.

EVSSL certificate is the global's digital certificate issued by institutions with browser vendors and together create the enhanced certificate, its main feature is the browser will

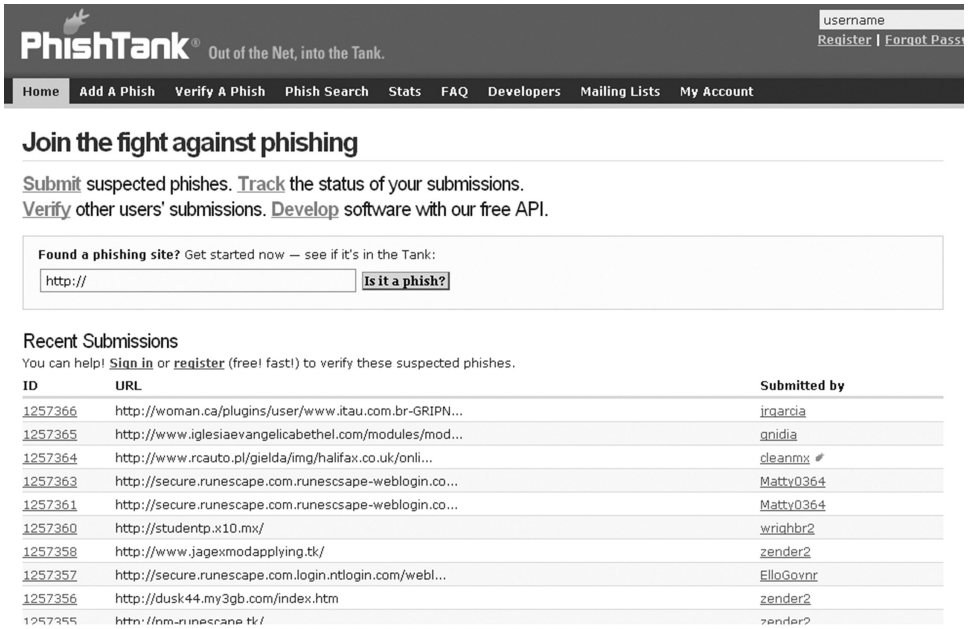

FIGURE 2.9 PhishTank list of malicious URLs.

give special treatment to the EVSSL certificate. EVSSL also follows the standard of X509 certificate and forward compatible with ordinary certificate. If the browser does not support EV mode, then we can make the EV certificate as a ordinary certificate; If the browser supports (need a new version of the browser) EV mode, it will be noted it in the address bar.

Therefore, if a website uses the EV SSL certificate, the address bar will turn green indicating that it is a legitimate site. This will help users in identifying and blocking phishing sites (Figure 2.13).

FIGURE 2.10 Effect of EV certificates on IE.

FIGURE 2.11 EV certificates in Firefox.

FIGURE 2.12 Ordinary certificate effects in IE.

FIGURE 2.13 Site with EV certificates in IE.

Although many users are not aware of this feature of browsers, the EV SSL certificate is widely used by websites. In the future, the popularity of EV SSL certificate authentication is expected to increase.

2.4 RAPID DEVELOPMENT OF BROWSER SECURITY

The scope of *security of browsers* is very wide, and today, the browser is still constantly updated with introduction of new security features.

In order to gain a competitive edge in the security field, Microsoft first introduced XSS Filter in IE8 to defend reflective XSS (cross-site scripting) attacks. XSS attacks are always

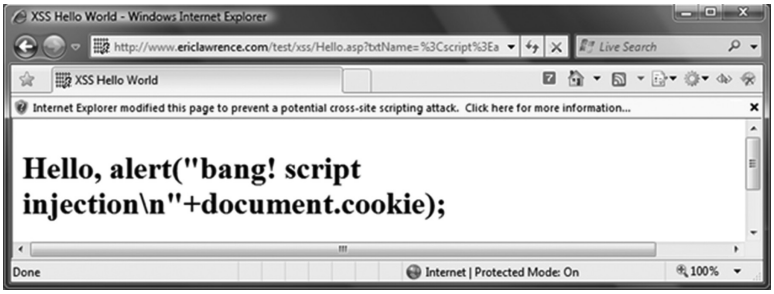

FIGURE 2.14 IE8 intercepted XSS attacks.

considered to happen due to application vulnerabilities at the server side, which should be patched in the code, and Microsoft first introduced this feature, making IE8 very unique in the security field.

When a user gets access to the URL containing an XSS attack script, IE will modify one of the key characters to prevent the attack from executing and will pop up a dialog box (Figure 2.14).

Some securities researchers decompiled IE8 executable files through reverse engineering and obtained the following rules:

```
{(v|(&[#()\[\].]x?0*((86)|(56)|(118)|(76));?))([\t]|(&[#()\[\].]
  x?0*(9|(13)|(10)|A|D);?))*(b|(&[#()\[\].]
  x?0*((66)|(42)|(98)|(62));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|
  A|D);?))*(s|(&[#()\[\].]x?0*((83)|(53)|(115)|(73));?))
  ([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(c|(&[#()\[\].]
  x?0*((67)|(43)|(99)|(63));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|
  A|D);?))*{(r|(&[#()\[\].]x?0*((82)|(52)|(114)|(72));?))}
  ([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(i|(&[#()\[\].]
  x?0*((73)|(49)|(105)|(69));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|
  A|D);?))*(p|(&[#()\[\].]x?0*((80)|(50)|(112)|(70));?))
  ([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(t|(&[#()\[\].]
  x?0*((84)|(54)|(116)|(74));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|
  A|D);?))*(:|(&[#()\[\].]x?0*((58)|(3A));?)).}

{(j|(&[#()\[\].]x?0*((74)|(4A)|(106)|(6A));?))([\t]|(&[#()\[\].]
  x?0*(9|(13)|(10)|A|D);?))*(a|(&[#()\[\].]
  x?0*((65)|(41)|(97)|(61));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|
  A|D);?))*(v|(&[#()\[\].]x?0*((86)|(56)|(118)|(76));?))
  ([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(a|(&[#()\[\].]
  x?0*((65)|(41)|(97)|(61));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|
  A|D);?))*(s|(&[#()\[\].]x?0*((83)|(53)|(115)|(73));?))
  ([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(c|(&[#()\[\].]
  x?0*((67)|(43)|(99)|(63));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|
  A|D);?))*{(r|(&[#()\[\].]x?0*((82)|(52)|(114)|(72));?))}
```

```
  ([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(i|(&[#()\[\].]
  x?0*((73)|(49)|(105)|(69));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)
  |A|D);?))*(p|(&[#()\[\].]x?0*((80)|(50)|(112)|(70));?))
  ([\t]|(&[#()\[\].]x?0*(9|(13)|(10)|A|D);?))*(t|(&[#()\[\].]
  x?0*((84)|(54)|(116)|(74));?))([\t]|(&[#()\[\].]x?0*(9|(13)|(10)
  |A|D);?))*(:|(&[#()\[\].]x?0*((58)|(3A));?)).}

{<st{y}le.*?>.*?((@[i\\])|(([:=]|(&[#()\[\].]x?0*((58)|(3A)|(61)|
  (3D));?)).*?([(\\]|(&[#()\[\].]x?0*((40)|(28)|(92)|(5C));?))))}

{[ /+\t\"\'`]st{y}le[ /+\t]*?=.*?([:=]|(&[#()\[\].]x?0*((58)|(3A)|
  (61)|(3D));?)).*?([(\\]|(&[#()\[\].]
  x?0*((40)|(28)|(92)|(5C));?))}

{<OB{J}ECT[ /+\t].*?((type)|(codetype)|(classid)|(code)|(data))
  [ /+\t]*=}
{<AP{P}LET[ /+\t].*?code[ /+\t]*=}
{[ /+\t\"\'`]data{s}rc[ +\t]*?=.}
{<BA{S}E[ /+\t].*?href[ /+\t]*=}
{<LI{N}K[ /+\t].*?href[ /+\t]*=}
{<ME{T}A[ /+\t].*?http-equiv[ /+\t]*=}
{<\?im{p}ort[ /+\t].*?implementation[ /+\t]*=}
{<EM{B}ED[ /+\t].*?SRC.*?=}
{[ /+\t\"\'`]{o}n\c\c\c+?[ +\t]*?=.}
{<.*[:]vmlf{r}ame.*?[ /+\t]*?src[ /+\t]*=}
{<[i]?f{r}ame.*?[ /+\t]*?src[ /+\t]*=}
{<is{i}ndex[ /+\t>]}
{<fo{r}m.*?>}
{<sc{r}ipt.*?[ /+\t]*?src[ /+\t]*=}
{<sc{r}ipt.*?>}
{[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).*?(((l|(\\u006C))(o|(\\
  u006F))({c}|(\\u00{6}3))(a|(\\u0061))(t|(\\u0074))(i|(\\u0069))
  (o|(\\u006F))(n|(\\u006E)))|((n|(\\u006E))(a|(\\u0061))
  ({m}|(\\u00{6}D))(e|(\\u0065)))).*?=}
{[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).+?{[\[]}.*?{[\]]}.*?=}
{[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).+?{[.]}.+?=}
{[\"\'].*?{\)}[ ]*(([^a-z0-9~_:\'\" ])|(in)).+?{\(}}
{[\"\'][ ]*(([^a-z0-9~_:\'\" ])|(in)).+?{\().*?{\}}}
```

These rules can capture the URL of XSS attacks, and other security products can learn from them.

Firefox also acted fast and launched a Content Security Policy (CSP), first proposed by security expert Robert Hanson. Its approach is to return an HTTP header from the server, in which security policies the page should comply with are described.

Because XSS attacks are unable to control the HTTP header in the absence of third-party plug-ins, this measure is feasible.

This custom syntax must be supported and implemented by browsers, and Firefox was the first browser to support this standard.

Using CSP by inserting an HTTP return header is as follows:

```
X-Content-Security-Policy: policy
```

The description of the policy is extremely flexible, such as

```
X-Content-Security-Policy: allow 'self' *.mydomain.com
```

Browsers will trust the contents from mydomain.com and its subdomain.

Another example:

```
X-Content-Security-Policy: allow 'self'; img-src *; media-src
  media1.com; script-src userscripts.example.com
```

Besides trusting their own sources, the browser will also load images from any domain, media files from media1.com, scripts from userscripts.example.com, and reject anything from other sources.

The concept of CSP design is undoubtedly good, but the rule configuration of CSP is complex. In the case of more pages, it becomes difficult to configure each page; maintenance cost also increases and promoting CSP becomes difficult.

Apart from these new security features, user experience for browsers is improving because of many *user-friendly* functions. But many programmers lack knowledge about these new features, which may cause some security risks.

For example, the address bar of the browser will respond differently toward the irregularity of a URL. The following URL will be properly parsed in IE:

```
www.google.com\abc
```

which will become

```
www.google.com/abc
```

The same thing happens in Chrome. "\" is changed to the standard "/".

But Firefox does not work this way: www.google.com\abc would be considered as an illegal address and will not be opened.

The same *user-friendly* functions can also be found in Firefox, IE, and Chrome. The following URL is very common:

```
www.google.com?abc
```

This becomes

```
www.google.com/?abc
```

Firefox can even recognize the following URLs:

```
[http://www.cnn.com]
[http://]www.cnn.com
[http://www].cnn.com
.....
```

However, if exploited by hackers to bypass the security software or security modules, these features will not be *user-friendly* any more.

Browser plug-ins also need to be considered as a threat to browser security. In recent years, abundant extensions and plug-ins have been the focus of reinforcing browser security.

Extensions and plug-ins greatly enriched the functionality of the browser, but security issues have also cropped up. Besides the loopholes plug-ins may have, a plug-in itself may be malicious. Extensions and plug-ins have higher privileges than the JavaScript page; for example, they can conduct some cross-domain network requests.

Sometimes, plug-ins might also contain malicious programs, such as the plug-ins named Trojan.PWS.ChromeInject.A, which is used to hack online banking passwords. It has two files:

```
"%ProgramFiles%\Mozilla Firefox\plugins\npbasic.dll"
"%ProgramFiles%\Mozilla Firefox\chrome\chrome\content\browser.js"
```

It will monitor all websites browsed in Firefox; when it identifies an online banking website, it will record the passwords used and then send them to a remote server. With new features come new challenges.

2.5 SUMMARY

The browser is an important entrance to the Internet, which has been increasingly valued by both offence and defense security personnel. In the past, when speaking of offence and defense, we paid more attention to server-side vulnerabilities, but right now, the scope of security research has covered all the aspects of the Internet, with the browser being the most important.

The security of browsers is based on the same-origin policy, so understanding the same-origin policy will help grasp the essence of browser security. In the current, rapidly developing trend of browsers, malicious URL detection, plug-ins, and other security issues will become increasingly important. Keeping up with the pace of browser development to study the security of browsers is what researchers need to take seriously.