

Whitepaper

# **SESSION RIDING**

## **A Widespread Vulnerability in Today's Web Applications**

Thomas Schreiber, SecureNet GmbH, Dec 2004

---

## TABLE OF CONTENTS

---

<b>1</b>	<b>Abstract</b> .....	<b>3</b>
<b>2</b>	<b>What is Session Riding?</b> .....	<b>3</b>
<b>3</b>	<b>How Session Riding Works</b> .....	<b>4</b>
	3.1 How Cookies and Session IDs work.....	4
	3.2 The Threat.....	5
	3.3 Where is the Vulnerability? .....	6
	3.4 Non-form-based Authentication Methods.....	6
	3.5 Summary .....	7
<b>4</b>	<b>What makes things even worse</b> .....	<b>7</b>
	4.1 Vulnerable GET-Requests .....	7
	4.1.1 The IMG-Tag .....	7
	4.1.2 The worst case .....	8
	4.1.3 IMG-Tags and Redirection.....	8
	4.1.4 Some observations .....	9
	4.2 Legal aspects.....	9
<b>5</b>	<b>Threat Scenarios</b> .....	<b>10</b>
	5.1 Administration Applications.....	10
	5.2 DSL-Routers .....	10
	5.3 Corporate Webmail and Sidestepping of Sender-ID .....	11
	5.4 Password Reset .....	11
	5.5 Custom Web Applications .....	11
	5.6 Intranets .....	11
	5.7 Single Sign-On .....	11
	5.8 WebDAV and other Http-Extensions .....	12
	5.9 One-time-Token and TANs .....	12
<b>6</b>	<b>Countermeasures</b> .....	<b>13</b>
	6.1 Solution 1: Use secrets .....	13
	6.2 Solution 2: Do URL-Rewriting.....	13
	6.3 What does not help .....	14
	6.3.1 Confirmation Pages .....	14
	6.3.2 POST instead of GET.....	15
	6.4 Workaround at the client side.....	15
<b>7</b>	<b>Conclusion</b> .....	<b>15</b>
<b>8</b>	<b>About SecureNet</b> .....	<b>16</b>
<b>9</b>	<b>References</b> .....	<b>16</b>

---

## 1 ABSTRACT

---

In this paper we describe an issue that was raised in 2001 under the name of Cross-Site Request Forgeries (CSRF) [1]. It seems, though, that it has been neglected by the software development and Web Application Security community, as it is not part of recent Web Application Security discussions, nor is it mentioned in OWASP's Top Ten [2] or the like. After having frequently observed this vulnerability in our Web Application Security assessments of custom Web applications, we started to examine various public Web applications and other browser-based applications:

- popular (commercial) Web sites
- popular browser-based console applications such as administration tools for databases, servers, etc.
- browser-based administration clients of hardware devices
- webmail sites and open source and commercial webmail solutions

We have found out that this vulnerability is present in many of those sites, services and products, some of which perform sensitive tasks. Actually, the list of affected companies contains well-known big players. Our analysis has led us to the conclusion that this vulnerability is the most widespread one in today's Web applications right after Cross-Site Scripting (XSS). Even worse, in some scenarios it has to be considered much more dangerous than XSS.

We feel that a concise description of this issue is necessary, along with a description of scenarios that highlight the danger to all browser-based applications that do not provide appropriate counter-measures, be it Intranet, Internet or console applications. In this paper, we explain this vulnerability in depth, show that it may be used unnoticed by the victim, describe potential threats, and finally give hints on how to make Web applications safe from such attacks.

We prefer to call this issue **Session Riding** which more figuratively illustrates what is going on.

---

## 2 WHAT IS SESSION RIDING?

---

In short: with Session Riding it is possible to send commands to a Web application on behalf of the targeted user by just sending this user an email or tricking him into visiting a (not per se malicious but) specially crafted website. Among the attacks that may be carried out by means of Session Riding are deleting user data, executing online transactions like bids or orders, sending spam, triggering commands inside an intranet from the Internet, changing system and network configurations, or even opening the firewall.

## 3 HOW SESSION RIDING WORKS

### 3.1 How Cookies and Session IDs work

In order to understand Session Riding let's first illustrate the main principle it is based on by having a look at cookies.

Cookies have a lot of subtle properties. The one that forms the basis for Session Riding is a very simple one. In short and a little simplified:

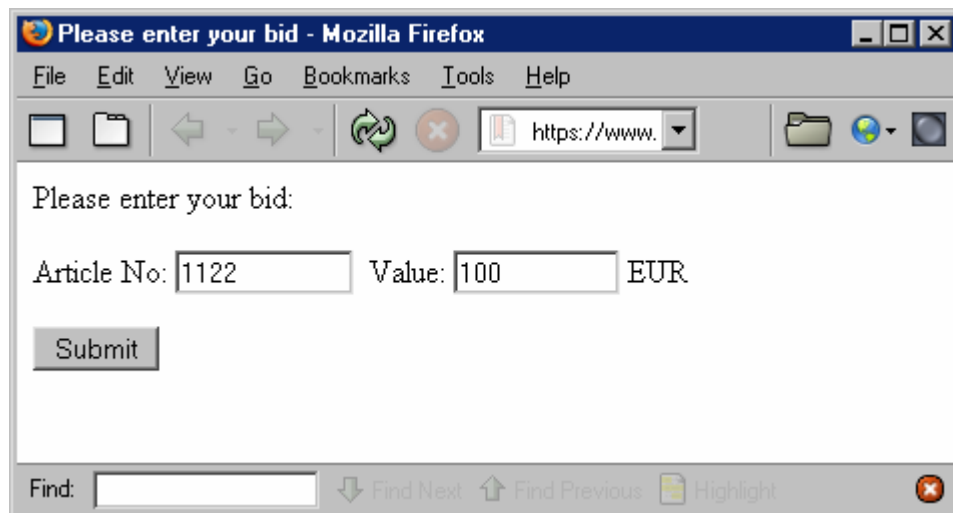
The browser sends a cookie that was set by some site A along with every further request to site A.

If this mechanism is looked at in more depth, more conditions have to be met: the path that is accessed must match the path the cookie was defined for, the lifetime of the cookie must not be expired, etc. We can ignore them for most of the following considerations, as they are not critical elements of the underlying principle.

Let's illustrate a typical use of cookies in the context of form based user authentication by an example: We look at a bidding application at <https://www.server.tld/myApp> where a buyer logs in and submits offers. Let's assume that this application uses cookies as the carrier for the session ID.

Suppose that the user has successfully logged in, so that a cookie is already set in his browser.

Then, he fills out the form to make his bid:



The screenshot shows a Mozilla Firefox browser window titled "Please enter your bid - Mozilla Firefox". The address bar displays "https://www.". The main content area contains the text "Please enter your bid:" followed by two input fields: "Article No:" with the value "1122" and "Value:" with the value "100", followed by the text "EUR". Below these fields is a "Submit" button. At the bottom of the browser window, there is a search bar with the text "Find:" and buttons for "Find Next", "Find Previous", and "Highlight".

When the user hits the Submit button, a request (in this case a GET request) like the one below is sent to the server:



The cookie `SESSIONID=123456789` is automatically added to the request. Thus, the application is able to recognize the user, confirm that he has authenticated before, and carry out the transaction based on the user's data, i.e. to submit a bid for an article with no. 1122 at a value of 100 EUR on behalf of the user who has previously logged in.

For those that are not so familiar with session management, let's point out one important thing concerning session IDs: after having logged in, the application recognizes the user solely by the session ID that the browser sends with the request. It does not ask for the user ID and password again. (As a consequence, if a valid session ID is acquired by another person, in general this person has access to the user's account in the application. Nevertheless, Session Riding has nothing to do with this so-called Session Hijacking.)

### 3.2 The Threat

Although it is very simple, let's derive the issue that is behind Session Riding step by step:

Where does the GET request that is sent in this example come from? It is generated by the browser after the user has selected the article, filled in his price and clicked on the Submit button, of course.

This is certainly the regular case, but is it technically the only possible way?

No, there are a few more ways: The user might as well have typed the link with all the parameters into the browser and hit Enter. He might have bookmarked the link before and selected it now. Or he might have found it on some other Web site and clicked on it. He even might have received an email with this link and clicked on it. In any of those cases, whenever the link is sent to the Web server – and the user has already logged into the application –, the browser supplies the Session cookie automatically and unavoidably. Thus, in any of those cases mentioned, the application received a perfectly correct bid and executed it in the context of the user. That's all!

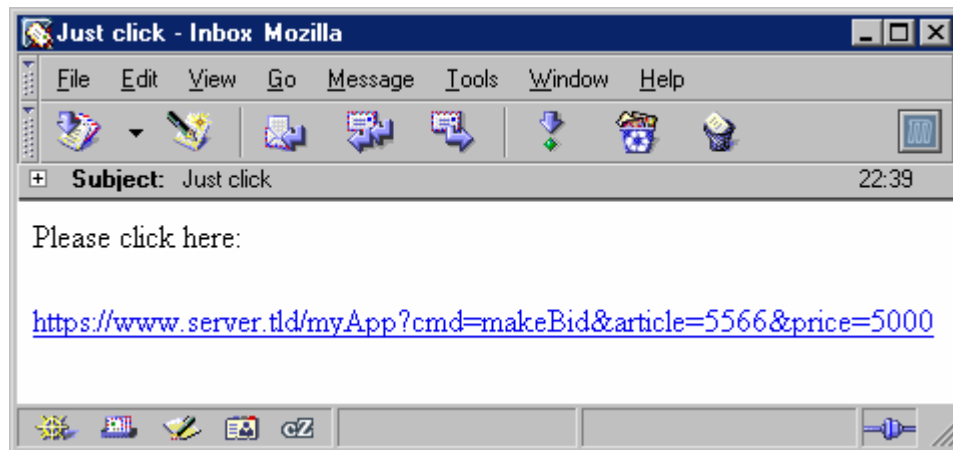
But what does this mean to Web Application Security?

To answer this, let's change our point of view and let's have a look onto this setting from an attacker's perspective. We assume that an attacker wants to push-up the price for an article. In order to achieve this, he only needs to:

- send the user (victim) an email with the link that contains the bid command (and convince him to click onto the link or otherwise provide for execution of the request)
- make sure that the user is logged in (i.e. a cookie with a valid session ID is still in the user's browser) when he executes the link.

We call the link that carries out the intended command the **effective link**, or more generally in case of a non-GET-request, the **effective request**.

Example of such an unmasked effective link:



If the attacker succeeds, the user will have made a bid for article no. 5566 at a price of 5,000 EUR.

The attacker abuses an existing session in the victims browser, he "rides" on it. Hence we refer to this threat as **Session Riding**.

### 3.3 Where is the Vulnerability?

What is wrong with the Web application, so that such an abuse is possible? The answer is: not much. The Web application implicitly assumes that any request that comes in expresses the will of the user, as it comes from the users browser. The application does not take into account that the request may as well have been foisted on the user by a third party.

One may argue that it is not the fault of the Web application but rather the fault of the immature Web technology. But with circumstances being the way they are, the Web application will have to handle this situation. Section 6 shows how to do so.

### 3.4 Non-form-based Authentication Methods

The principle that forms the basis of Session Riding is not restricted to cookies. Basic Authentication is subject to the same problem: once a login is established, the browser automatically supplies the authentication credentials with every further request automatically.

We have not tested Digest Authentication, NTLM Authentication and client certificates yet. But as all those techniques work along the same principle, we suppose that they are subject to Session Riding as well.

### 3.5 Summary

To summarize, these are the prerequisites that make a Web application vulnerable to Session Riding:

- Session Information is carried by some means that is automatically injected into the request by the browser. Cookies and Basic Authentication are such means.
- Any information that is used in the effective request may be constructed from knowledge that is available to a third person.

Additionally, in order for an attacker to carry out an actual attack, he has to

- make sure that the targeted user is logged into the application when the effective request is executed
- be able to get the knowledge to construct the effective request.

The last point mentioned is often feasible: in public Web applications the attacker may register for an account; Open Source console applications he may download and install; hardware devices or commercial applications he may buy; for intranet or restricted Web applications he may use his own account to study the application, etc.

## 4 WHAT MAKES THINGS EVEN WORSE

---

### 4.1 Vulnerable GET-Requests

In the case where an application makes use of GET requests, Session Riding can be accomplished without any user interaction and without JavaScript. The only thing the user must do is *open* a manipulated Web page, nothing more. The page with the malicious content (i.e. the effective link) may even be on some popular website that the user trusts so that he may accidentally initiate the attack when he browses this site. At least, there is little the user can do to prevent the attack from being carried out once the poisoned page has been opened, as the effective link is executed automatically.

#### 4.1.1 The IMG-Tag

The key for this almost unavoidable and unnoticeable form of Session Riding are those tags that automatically trigger a request of the browser. The IMG-Tag is of this kind<sup>1</sup>. The browser that opens a page at `http://www.somesite.tls/` with this content

---

<sup>1</sup> In theory the user may disable the loading of images, in practice most websites are not usable without image display.

```
<HTML><BODY>
  Can you see this picture:
  <IMG SRC="https://b2b.company.tld/webapp/delete?item=123"
</BODY></HTML>
```

accesses `b2b.company.tld` to fetch the alleged picture `/webapp/delete?item=123`. The fact that the targeted resource is not a picture at all does neither prevent the browser from sending the authentication credentials or cookies with the request, nor does it prevent the (malicious) application from executing the request. The reason for the former is that the browser cannot check at request time whether a URL is appropriate for the invoking tag, and it sends the credentials merely based on host and path. The reason for the latter is that there is no indication in the http protocol that tells the Web server that the request has been initiated by a certain HTML tag, in our case an image tag. The fact that the result of the request is some HTML code only leads to a broken image symbol in the displayed page, nothing else. What's important: the effective request is perfectly executed.

Any website that allows the integration of HTML (or at least images served by external websites) may be used as a carrier for the malicious command. Suppose the image link is part of the offer of a popular auction site. The attacker must now only trick the user into looking at this offer, what is perhaps easier than to betray him to click on an unknown link. Or he starts a mass attack where he induces a lot of users to look at his offer – perhaps by offering a very demanded item – reckoning that there are a certain amount of users among them that are currently logged into the application that he targets with his malicious link.

#### 4.1.2 The worst case

The worst case in this scenario is probably the situation where the application that is targeted and the application that is the carrier are the same. If so, any user that looks at the page is inevitably logged into it in that very moment and thus has no chance to escape the attack. During our analysis we found popular sensitive applications with this behaviour.

#### 4.1.3 IMG-Tags and Redirection

Browsers do follow redirects in IMG access. Thus an attacker may even put some sort of logic into his Session Riding attack. Instead of poisoning a website with the direct access to the target site, he redirects it from his own site. In the bidding example above, the link might be:

```
<HTML><BODY>
  Can you see this picture:
  <IMG SRC="https://attakersite.tld/trigger"
</BODY></HTML>
```

which is answered by the server with a 302-Redirect to

```
https://www.server.tld/myApp?cmd=makeBid&article=5566&price=5000
```

with some price. The next one who accesses this link is redirected to the same link with some higher price:

```
https://www.server.tld/myApp?cmd=makeBid&article=5566&price=5100
```



and so on. The logic is provided by the attacker's application (<https://attakersite.tld/trigger> in this example) and can be tailored to each target application.

#### 4.1.4 Some observations

At the time of writing, the latest version of the below mentioned products exhibit the following behaviour regarding images, cookies and authentication credentials:

If Outlook displays (both in open and preview mode) an HTML email and image display is not switched off (in the default configuration it isn't), it sends existing cookies in Internet Explorer along with the image request if those cookies were set by the `Set-Cookie-Header`. It does not send cookies that have been set by JavaScript, and it does not send Basic Authentication credentials either. Thus, in the first case a Session Riding attack is executed by just looking at an email in Outlook's preview mode.

Opera, where browser and email reader are tightly integrated as well, sends all cookies and as well basic authentication credentials with every image request of an HTML email.

Firefox and Thunderbird seem to not support Session Riding directly out of the email.

## 4.2 Legal aspects

Part of the nature of Session Riding is the fact, that the action is carried out from the PC of the targeted user at the time the user is working at his PC. Thus Session Riding is a very serious issue in the field of proof of evidence.

Consider this: a transaction of certain importance has been carried out by some person A. Let's say, it's the submission of a legally binding bid. In fact, but unnoticed by A, it has been carried out by an attacker via Session Riding. The seller, B, now sends out the product and the invoice. But A clearly opposes as he has not conducted this transaction. He accuses B of fraud as B does of him. As B insists on the execution of the deal, the dispute amounts to a lawsuit. In the course of hearing of evidence, the logfiles of the auction service provider are checked for the IP address of the sender in the case in question. Also, the Internet provider is asked to reveal details about the person who has used this address at the time in question. The conclusion will of course be that the transaction was carried out from A's PC. To make the case a bit tougher, a witness testifies that A was working at his PC at the time in question. Would a judge accept this as proof of evidence and rule A guilty? Hopefully not, as a Session Riding vulnerability in the application might have been used to initiate the transaction from a third party, and A has committed nothing more than opening an email while being logged into the target application. Interestingly, given this scenario, a Session Riding attack must be considered worse than stealing a password: when the attacker logs in with stolen credentials he typically uses a different IP address from his victim, and thus no evidence exists to prove that the order has been placed from the victim's PC.

The wide field of human weaknesses yields a more obvious scenario: consider the employee who wants to bring a colleague he dislikes into trouble: If the company

uses a vulnerable webmail product, it is easy to place a well timed email that, when opened by the colleague, sends a compromising email (e.g. containing offending or even criminal content) to the manager or so. The carrier email may even be sent anonymously from the Internet. In a corporate environment it is normally easy to track down the actual sender, find the compromising email in his outbox and obtain evidence that the person was working at his PC at the time the email was sent.

## 5 THREAT SCENARIOS

---

After having explained the main technique of Session Riding with a basic example, let's get an impression of the danger of this kind of flaw. Consider the following scenarios:

### 5.1 Administration Applications

Nowadays, most hardware devices in a network (e.g. network appliances) are configured and controlled by an HTML-client. Often Basic Authentication is used, thus those devices are potentially vulnerable.

If the administrator of such a vulnerable device is logged into the device and opens a malicious website or email with the same browser, he is subject to attacks. Imagine the worst case, where the administrator of a large company is constantly logged into his firewall appliance because he needs to configure changes throughout the day. A malicious link executing unnoticed by the administrator may open the firewall.

The situation is similar with browser-based admin consoles for software like Web and application servers, databases, ldap servers, etc.

Naturally, Web applications of this kind have a certain criticality. They should absolutely not be subject to Session Riding. What we have found is that Session Riding is actually very common in this area. Among the problematic cases are popular open source software tools as well as widely used products of well known manufacturers.

### 5.2 DSL-Routers

Routers for private use – e.g. those popular DSL routers – come with a pre-configured IP address (e.g. 192.168.0.1) for the LAN interface that most users do not change. So the host-part of the URL is known in most cases. The rest is easily constructed if you have access to the same hardware model. E.g.

```
http://192.168.0.1/enable_remote_management=true&ips_allowed=*&mgmt-port=8080
```

switches on remote management on port 8080 from every computer on the Internet.

### 5.3 Corporate Webmail and Sidestepping of Sender-ID

Many companies give their employees remote access to their email account by means of a webmail interface. There are quite a few such products, commercial and Open Source, around. Imagine the email-send function were vulnerable to Session Riding: as the user is necessarily logged into his webmail account when he reads an email, the session is open in the moment of accessing the malicious mail. So it is easy for a spammer to transport a spam mail piggybacked on an email to the victim. When the user triggers the malicious payload, the email is sent via the company's email gateway. Thus, internal users are easily accessible. So are Internet users that rely on blacklist filters as the mail comes from an unlisted relay. And even the upcoming Sender-ID technique is elegantly sidestepped as the sending user and the sending server do match.

And then there is the legal aspect: how will a such fooled user proof that an email that was verifiably sent from his PC was actually not sent by him? Paragraph 4.2 elaborates more on this aspect of Session Riding.

### 5.4 Password Reset

Most prominent Web applications today conduct a safe password reset: before changing the user's password, they ask for the old one and thus are safe from Session Riding. Some do not. If those applications use the affected authentication methods, an attacker may change the password to his will and then login, using the victim's identity.

### 5.5 Custom Web Applications

Based on our experiences with Web Application Security assessments of custom Web applications we conclude that there are countless applications out on the Web that are vulnerable to Session Riding. To name just a few possible threats: remove the competitor's offer from the customer's bidding platform; place expensive orders on behalf of the competitor at some trading portal; delete users, add new users and change passwords; trigger emails that forward sensitive content to the own mailbox; attack the intranet (see below), etc.

### 5.6 Intranets

Intranet Applications are of course equally affected. What makes the issue probably more threatening is the fact that in a corporate environment more insider knowledge is openly available, like: the time when some person is logged into an application, the person's email address, details about the functioning of an application. So the success rate of an inside attacker may be much higher. As the effective request may be inside an Internet page, the inside attacker may mount the attack anonymously and riskless from the outside.

### 5.7 Single Sign-On

In a Single Sign-On environment a user logs in to one central, specially secured login authority and from then on is able to use various applications without further logging in to them individually. Single Sign-On is becoming more and more common in enterprise environments.

If cookies are the transportation mechanism of the login-token and no counter-measures are taken, then the Single Sign-On solution is vulnerable to Session Riding. In such an environment its impact is particularly strong:

- Session lifetimes tend to be much longer than those of normal web applications
- The user is not able to selectively login to only a certain application in order to limit impact
- Even if a user only actively uses certain applications he is threatened by Session Riding to all other applications for which he has an account or access to.

## 5.8 WebDAV and other Http-Extensions

There are several extensions to http, most notably WebDAV. WebDAV allows the manipulation of remote file systems (including uploading, i.e. writing files) via http. It is becoming increasingly popular, for example with Web space providers. Other extensions are proprietary. As an example, some companies use such extensions for the communication between their Java administration clients and the hardware or software devices and appliances that are equipped with a custom embedded Web server.

Although browsers do not support those proprietary extensions nor WebDAV out of the box, it may not be concluded that WebDAV is safe from Session Riding, nor are those proprietary extensions generally safe.

The reason for this is that JavaScript functions like `document.form.submit()` and ActiveX-Objects like `Microsoft.XMLHTTP` allow the submission of non-standard http requests to some extent. If ActiveX is enabled in the client browser, a manipulated website can mount a WebDAV PUT (upload) or a DELETE, for example. For Session Riding to be carried out here – leading to the upload of a file to the Web server, for example – , it is sufficient that the user is logged into the WebDAV-Server with his browser at the moment he views the malicious website.

Even custom Java Clients are vulnerable if they use the DOM of the browser. If so, the browser is in the same context as the Java client regarding cookies and authentication credentials that have been set by the Java client. Thus, the browser sends those credentials with every request to the host in question as well.

## 5.9 One-time-Token and TANs

In Germany, online banking transactions are secured by a one time token. It is called the TAN (Transaction Number). That is, with every transaction a parameter that is only known to the user is transmitted to the Web application. Thus, it is safe against Session Riding. But nevertheless, there is an effect on security. As the level of security is the combined protection of password and one-time token, an online banking application that fulfils the preconditions of Session Riding weakens this security, as an attacker now has a chance to initiate a money transfer if he only knows a TAN. The conclusion: applications that secure transactions by means like one-time tokens have to be aware of Session Riding as well.

## 6 COUNTERMEASURES

---

### 6.1 Solution 1: Use secrets

The (general) solution is simple – in theory, at least. In practice it may result in substantial extra work. This holds especially true if the countermeasure is retrofitted into an existing Web application.

To make a Web application safe against Session Riding, introduce a secret (aka hash, token) and put it into every link and every form, at least into every form that is sensitive. The secret is generated after the user has logged in. It is stored in the server-side session. When receiving an http request that has been tagged with this information, compare the sent secret with the one stored in the session. If it is missing in the request or the two are not identical, stop processing the request and invalidate the session.

The result is, that instead of sending this

```
https://www.server.tld./myApp?cmd=makeBid&article=5566&price=5000
```

to place an order, the browser sends this

```
https://www.server.tld./myApp?cmd=makeBid&article=5566&price=5000&secret=ko2498geroihu78idsf78
```

to the Web application. As the attacker cannot know this secret he is not able to inject such a (valid) link.

An application or at least a certain dialog sequence or functionality of an application is not vulnerable, if it already carries a particular parameter that is not guessable from the outside. Sometimes one observes random 'LoginIDs' or 'request numbers' in applications in addition to the session ID. If they are truly random, there is probably no need for an additional secret.

What certainly will not remedy an application from session hijacking is to put the secret into a cookie instead of into a parameter, as it is then automatically sent along with the session ID.

You should generally also not use the session ID as secret. Although it raises the bar on exploiting Session Riding substantially, any XSS weakness in the application or site (depending on the cookie definition) would allow an attacker to construct the secret into the Session Riding code. One might argue, that with XSS we are on the ground of Session Hijacking, so why should an attacker do Session Riding if he can even do Session Hijacking. But as paragraph 4.2 above shows, Session Riding may pose more harm than Session Hijacking.

### 6.2 Solution 2: Do URL-Rewriting

URL rewriting is safe from Session Riding as it lacks the critical automatism of information being sent automatically. So the second solution for a subset of the problem cases is as simple as this: when doing form-based authentication in the application, use URL rewriting instead of cookies.

For some applications it is trivial to switch to URL rewriting as most application servers have a switch to do so transparently for the application. But for most applications it is probably a lot of work as every link has to be checked or modified individually.

At the risk of initiating another discussion on the security of cookies versus URL rewriting for session management, and neglecting the fact that cookies are much more convenient from the software engineering point of view, we state this: not only because of Session Riding, we consider URL rewriting for session tracking the safer method. The Session Riding threat makes this advantage even stronger.

## 6.3 What does not help

### 6.3.1 Confirmation Pages

A "Do you really want to delete/order/change/... this?" step before executing the command normally does not help.

Consider the deletion of some item in some Web application. When the user selects the delete command, the browser sends something like this to the Web server:

(1) `https://b2b.company.tld/webapp/delete?item=123`

The response page confirms if he really wants to do this. When he responds affirmatively, some request like this results:

(2a) `https://b2b.company.tld/webapp/delete?item=123&checked=1`

or

(2b) `https://b2b.company.tld/webapp/realdelete?item=123`

As this second request contains all the information necessary to carry out the deletion, there is no need for an attacker to go through (1). An attacker would skip step (1) and would directly hand over URL (2a) or (2b) to the victim. So this poses no barrier against Session Riding.

Instead of using options (2a) or (2b), the application might store the identity and state of the item that is intended for deletion in the session and not transport it back to the client. In that case, the delete command would relate to the last item that has been selected. The corresponding URL might look like this:

(2c) `https://b2b.company.tld/webapp/delete?checked=1`

It seems that the attacker can only trigger the deletion with (2c) if the item is already in the delete state, thus it would be very unlikely that he would be able to carry out a specific attack (as it would be unlikely that the victim would just be in the process of deleting the item with the same ID that the attacker uses, right at the moment when the malicious URL comes in). But in fact, the attacker may put both URLs, (1) and (2c), into his carrier. With the help of JavaScript he can delay (2c) a little so that the state is settled properly when it executes. So this as well generally is no real protection against Session Riding.

### 6.3.2 POST instead of GET

All examples in this paper are carried out using GET requests. That does not mean that applications using POST are secure. It is only that it's somewhat easier to illustrate this issue with GET and that there are more situations where a user may be caught if GET is accepted by the application. Especially the IMG-Tag trick is only possible with GET. A POST may as well be carried out without user interaction if JavaScript is enabled. Otherwise, the attacker must trick the user into clicking onto some button.

(Nevertheless Session Riding is another example where using POST instead of GET raises the bar for a successful exploits a little higher.)

### 6.4 Workaround at the client side

What can users do to be safe from vulnerable applications?

**Administrators:** In the case of the full-time administrators from above that use a Web console to administer the application, we suggest: if you are not absolutely sure that the Web-enabled client of your device is Session Riding-safe, do not use it. If you have to use it, ensure that the browser you use for the administration task is not the standard browser. But best, do not use this workstation for anything else than controlling the device, do not open other websites and do not read your email on this machine. And, of course, disable Java Script and all Active Scripting if possible.

**Regular Users:** The situation for the general Internet user is much more difficult. Perhaps the best he can do is to use a second browser – i.e. the one that is not configured as the standard browser which automatically pops up when a link is triggered – for the more sensitive applications like webmail, Internet banking, auctions, shopping, etc. Additionally, he should immediately log out of any online application after finishing his transactions, and of course he should not click on links that leave the application. But under practical conditions he can do nothing to be absolutely safe at the moment. If one of those applications is vulnerable itself in the sense described in 4.1.2, he can do absolutely nothing practicable to prevent himself from being attacked.

## 7 CONCLUSION

---

Hopefully we have provided enough insight and examples to make clear how dangerous Session Riding is. It looks like a big quantity of browser based applications are effected these days. Anyone responsible for the security of a site should check all applications for the presence of Session Riding and remedy it. Open Source and commercial software developers should do the same for their browser controlled applications and devices.

In our opinion, Session Riding is largely an issue of immature Web technology. At least browsers and frameworks like the Servlet technology should provide built-in measures that result in safe applications. It should not be the responsibility of Web application developers to compensate for the lack thereof.

As Session Riding is so powerful and easy to conduct – it even has a certain "fun factor" to the evil-minded – it is likely that it may soon become a widely exploited issue.

Any Web Application should implement a measure against Session Riding. The best way to do so generally is to carry a *secret* in the URL.

---

## 8 ABOUT SECURENET

---

The Munich, Germany, based SecureNet GmbH develops custom business-to-business Web applications and is specialised in the security of Web applications. SecureNet provides Web Application Security Consulting, Audits and Penetration Tests.

---

## 9 REFERENCES

---

- [1] Cross-Site Request Forgeries, <http://www.securityfocus.com/archive/1/191390>
- [2] The Open Web Application Security Project, <http://www.owasp.org>
- [3] CERT® Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests, <http://www.cert.org/advisories/CA-2000-02.html>



Dec 2004

SecureNet GmbH  
Münchner Technologiezentrum - Frankfurter Ring 193a - D-80807 München, Germany  
[www.securenet.de](http://www.securenet.de) - [info@securenet.de](mailto:info@securenet.de) - Phone +49/89/32133-600