Cross-Site Scripting & *Cross-Site Request Forgery*

Johnny Souza 04/85942

Universidade de Brasília
Departamento de Ciência da Computação
Segurança de Dados, 1/2009

15 de junho de 2009

Resumo

Este trabalho aborda a história e os mecanismos básicos de ataques *Cross-Site Scripting* e *Cross-Site Request Forgery* por meio de exemplos com código fonte funcional.

1 Introdução

As definições de *Cross-Site Scripting* (XSS) e *Cross-Site Request Forgery* (CSRF ou XSRF) são frequentemente confundidas e às vezes tratados até como o mesmo tipo de ataque [8].

Um aspecto muito importante para a diferenciação correta destes dois tipos de ataque é a confiança. Quanto um usuário acessa um site disponível através da Internet, este usuário confia que todo o conteúdo recebido é originário (e legítimo) do servidor no qual está hospedado o site acessado. A confiança também é observada na via contrária, ou seja, o servidor confia que foi o usuário que fez a requisição intencionalmente e a atende.

Os ataques XSS tiram proveito da confiança que o usuário tem no servidor[1, 8] e atuam inserindo conteúdo alheio à página esperada, esse conteúdo pode ir desde informações falsas, até a códigos HTML e/ou JavaScript que podem ser usados para realizar ataques CSRF[1].

Já os ataques CSRF se apóiam na confiança que o servidor de um site, ou aplicação Web, tem de que é o usuário que efetivamente está realizando as requisições e que o faz conscientemente[1, 8]. Atuam através de códigos maliciosos, forjando requisições e simulando navegações pelo site[1] sem que o usuário perceba, de forma silenciosa[6].

Ao contrário do que pode parecer, o CSRF não é um ataque que burla a autenticação do usuário. Ele ocorre após a autenticação fazendo requisições ao site como se fosse o usuário, porém sem que o usuário perceba.

2 Histórico

2.1 *The Confused Deputy*

Em 1988, Norm Hardy publicou um artigo[7] abordando uma falha de confiança na camada de aplicação[1]. O artigo [7] descreve um fato ocorrido por volta de 1977 em uma companhia de *time sharing*, a TymShare.

Essa companhia possuía um sistema de cobrança baseado no tempo de uso de seu compilador que era compartilhado entre vários usuários. O tempo de uso por usuário era registrado, para fins de faturamento, em um arquivo ao qual somente o processo do compilador tinha acesso. O controle de acesso a este arquivo era feito pelo sistema operacional em uso.

Uma da funcionalidades desse compilador era a possibilidade de especificar um arquivo no qual o compilador escrevia informações de *debug* sobre o programa compilado. Fazendo uso desse mecanismo de *debug*, algum usuário que conhecia o *path* do arquivo que armazenava os registros de tempo de uso, direcionou a saída das informações de *debug* para esse arquivo, sobrescrevendo todas as informações que seriam usadas para cobrança dos clientes.

Como o processo que estava acessando o arquivo com os registros de tempo de uso é o processo do compilador o sistema operacional, confuso, não bloqueou o acesso indevido.

2.2 *Cross-Site Request Forgery*

Em 2000, foi publicada uma descrição de como o servidor de aplicações ZOPE era afetado por um problema do tipo *confused-deputy* que ocorria através da Web, onde é possível induzir o navegador a fazer requisições a um site no qual o usuário havia se autenticado previamente[6].

Em 2001, Peter W. utilizou o termo *Cross-Site Request Forgery* para descrever um ataque muito semelhante ao descrito em [6] que utilizava o atributo SRC das *tags* IMG para fazer as requisições maliciosas[10].

É um tema que não é frequentemente abordado em artigos acadêmicos e publicações de tecnologia, porém é considerado por [11] como um gigante adormecido dentre os *bugs* da Internet.

2.3 *Cross-Site Scripting*

O aparecimento das vulnerabilidades do tipo *Cross-Site Scripting* remete para 1996 durante o inicio da expansão da Internet, sites divertidos usando

frames HTML e o surgimento do *e-commerce* e da linguagem JavaScript. O JavaScript trouxe muitas mudanças a Web, proporcionando aos usuários interfaces interativas e aos *hackers* um novo mundo de possibilidade inexploradas. [4]

A primeira, e mais simples, forma de exploração foi a possibilidade de códigos JavaScripts serem capazes de ultrapassarem a fronteira do *frame* em que estão contidos e poderem acessar informações de outros *frames*. Dessa forma, *scripts* de um site podem atuar sobre informações de outro site, acessar dados digitados em formulários (inclusive senhas) e roubar *cookies*. Esse fato foi divulgado pela imprensa como uma vulnerabilidade dos navegadores [4].

Em Janeiro de 2000, Microsoft, CERT, Apache e outros fornecedores de software se reuniram em Washington para discutir o conceito desse tipo de ataque[4]. Como resultado dessa reunião foram publicados os artigos [3], [5] e [9] nos quais o termo *Cross-Site Scripting* é definido em comum acordo para designar essa vulnerabilidade, é apresentado, também, uma descrição detalhada do problema incluindo até exemplos de código fonte. Inicialmente *Cross-Site Scripting* passou a ser conhecido pela sigla CSS, porém isso causou grande confusão com o recém criado *Cascading Style Sheets* (também CSS) até que alguém, nos primeiros anos da década de 2000, sugeriu o acrônimo XSS para evitar confusões[4].

A grande maioria dos especialistas em seguranças e desenvolvedores não davam muita atenção ao XSS. Enquanto se ocupavam com *firewalls* e *Secure Socket Layer* (SSL), pensavam que JavaScript (que possibilita o XSS) era uma linguagem de programação para "brincadeiras". Como se acreditava que os danos não poderiam ser grandes, não havia motivo para se preocupar. Até que em outubro de 2005, o Samy Worm derrubou o MySpace[4], uma das redes sociais mais populares da época.

3 Técnicas

3.1 CSRF

A figura 1 mostra o esquema básico de ataques CSRF.

Como precondição para que um ataque de CSRF seja bem sucedido, o usuário deve previamente ter efetuado *login* em um site confiável que faça autenticação implícita[1]. O sucesso do ataque não depende do tipo de autenticação, podendo ser autenticação básica HTTP ou utilizando *cookies* com ou sem uso de SSL[11, 6]. Quando um usuário legítimo (A) apresenta suas credenciais de identificação perante um site legítimo (B), o site passa a confiar que todas as requisições feitas através do mesmo navegador utilizado por A são legítimas[11].

[1] Autenticação explícita requer que o usuário informe o usuário e senha a cada requisição, o que causa grandes danos à usualididade

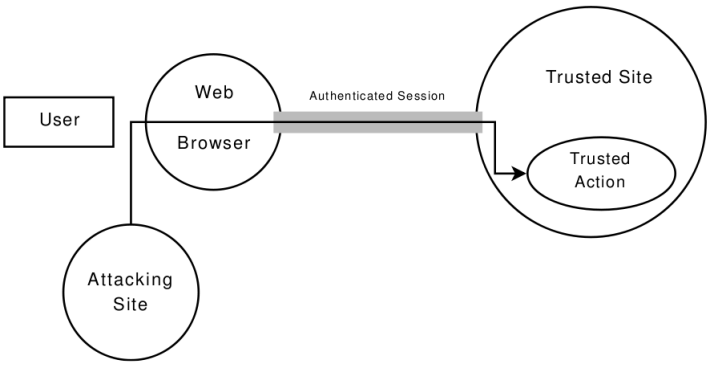

Figura 1: Ataque CSRF

Já com a sessão autenticada, se o usuário visitar um Web site mal intencionado (M), este site pode induzir o navegador de A a fazer uma requisição a B sem que A tenha consciência disso, forjando uma requisição. Ao fazer essa requisição o navegador de A utiliza a sessão autenticada previamente aberta, inclusive criptografando a requisição caso SSL esteja em uso.

Dessa forma, o navegador de A faz uma requisição forjada, e o site B a atende como se ela fosse legítima e do interesse de A. Em ataques CSRF, tudo parece legitimo porque tudo efetivamente é legítimo, exceto a intenção[8].

Os meios comumente utilizados para forjar uma requisição são recursos do HTML (*tags IMG*, por exemplo) ou de JavaScript. No exemplo retirado de [11], é descrito um serviço de *WebMail* (localizado em B) no qual A pode enviar mensagens de email através do formulário:

```
1  <form action="http://example.com/send_email.htm"
       method="GET">
     Recipient's Email address: <input type="text" name="to">
     Subject: <input type="text" name="subject">
     Message: <textarea name="msg"></textarea>
6    <input type="submit" value="Send Email">
   </form>
```

Listing 1: Formulário de envio de email

Como o formulário utiliza o método GET, quando o usuário A clica em *Send Email*, o navegador codifica os campos do formulário e faz uma requisição direta à URL:

```
http://example.com/send_email.htm?to=carol%40example.com&
subject=hello&msg=What%27s+the+status+of+that+proposal%3F
```

Sabendo desses detalhes, M pode colocar em uma de suas páginas uma tag IMG como a seguinte:

```
<img src="http://example.com/send_email.htm?
to=mallory%40example.com&subject=Hi&msg=My+
email+address+has+been+stolen">
```

Listing 2: Código malicioso

Quando A acessar essa página, o navegador irá requisitar a B a URL especificada no atributo SRC da *tag* IMG com o intuito de obter uma imagem para mostrar para A, porém só será possível saber que não há uma imagem nessa URL quando B enviar uma reposta, ou seja depois que a ação já foi executada (neste caso o envio de um email) sem o conhecimento e concordância do usuário A.

Caso o servidor só aceite requisições com dados enviados via POST, é necessário utilizar JavaScript para forjar um POST, apenas aumenta um pouco a dificuldade, mas não impede o ataque. Um fato importante de mencionar é que grande parte dos servidores aceitam tanto requisições POST quanto GET,ou seja, mesmo que o formulário utilize o método POST, o ataque pode ser feito utilizando o método GET.

3.2 XSS

A maioria dos ataques XSS tem como mecanismo básico o envio de código HTML ou JavaScript em campos de entrada de formulários que não são validados. O campo mais utilizado para este fim é o campo de busca em sites.

No exemplo abaixo, retirado de [4], é demonstrado um ataque capaz de roubar um *cookie* do usuário.

Suponha um sistema de *e-commerce* que utilize autenticação por *cookie*. Para efetuar compras nesse site, os usuários realizam autenticação mediante apresentação de nome de usuário e senha e recebem um *cookie* com o ID da sessão.

Nesse site existe um formulário de busca de produtos que utiliza o método GET e não validada a entrada dos usuários. Após um usuário entrar um termo de busca, na página de resposta é exibido o termo utilizado na busca e os possíveis resultados.

Como já foi mostrado anteriormente, é uma requisição feita através de um formulário é codificada em uma URL. Dessa forma, uma URL 'preparada' como a mostrada baixo é capaz de simular uma requisição.

```
http://e-commerce/search?q=<SCRIPT>var+img=new+Image();
img.src="http://hacker/"%20+%20document.cookie;</SCRIPT>
```

Como resposta, o servidor envia uma página com o resultado da busca contendo o seguinte trecho de código:

```
<p>Sorry, no search results found for
```

```
2 "<SCRIPT> var img=new Image();
  img.src="http://hacker/" + document.cookie;
  </SCRIPT>"</p>
```

Listing 3: Resultado de busca e roubo de *cookie*

O código acima possui um trecho em JavaScript que, quando é executado, envia para o computador de nome *hacker* os *cookies* referentes a este site, inclusive o de autenticação.

Como o *cookie* solicitado é de fato da página sendo exibida, o navegador não faz nenhum tipo de restrição de acesso ao mesmo, permitindo que o código malicioso envie a informação para *hacker*.

Para efetivar esse ataque, é preciso divulgar o *link* para essa URL por email, mensagem instantânea ou site na Internet. Como é um *link* para um site "confiável" e parece ser inofensivo (principalmente para usuários desatentos), com um pouco de engenharia social, será clicado facilmente. Caso algum desses usuários possuam uma sessão autenticada no momento em que clicam no link, terá o ID da sua sessão roubado.

Outra forma de XSS, que é conhecida como persistente, é mais simples e consiste de enviar código malicioso que é salvo pelo site vítima e exibido posteriormente para outros usuários. Isso pode ser feito em comentários de blogs, mensagens de email e mensagens em fóruns, por exemplo.

3.3 XSS + CSRF

"*Buy one XSS, get a CSRF for free*" – Johann Hartmann *apud* [4]

Os dois ataque são frequentemente utilizados em conjunto, por exemplo, pode-se inserir código (XSS) que faça uma requisição (CSRF). Suponha um sistema de fórum vulnerável, no qual um usuário mal intencionado envie uma mensagem e nela inclua código capaz de forjar uma requisição para excluir a conta de um usuário, quando os usuários vítimas acessarem essa mensagem, terão suas contas apagadas.

Outra possibilidade é explorar a vulnerabilidade de XSS de um site conhecido como ataque precedente a um ataque de CSRF em outro site, isso além de aumentar o número de vítimas, dificulta o rastreamento do autor do ataque.

4 Prevenção

Para o usuário final é muito difícil, senão impossível, proteger a si mesmo de ataques XSS enquanto estiver *on-line* mesmo com todas as atualizações e configurações de segurança[8]. Uma ação efetiva é desabilitar o JavaScript, porém atualmente isso é impraticável, há um número muito grande de sites que fazem uso intensivo de JavaScript, como sistemas em Ajax.

A proteção contra XSS deve ser feita do lado do servidor, com o desenvolvimento de aplicações que implementem técnicas eficientes de filtragem de formulário e parâmetros fornecidos via URL.

Os ataques de CSRF são de simples diagnóstico, simples exploração e simples solução. Eles existem porque os desenvolvedores não são educados sobre a causa e seriedade de ataques CSRF.[11]

A proteção contra CSRF pode ser feita efetivamente do lado do servidor, e o usuário pode tomar uma série de medidas para proteger a si mesmo de muitos tipos de ataques CSRF mesmo que o site não tenha a devida proteção. [11]

As técnicas de proteção contra CSRF são bastante simples e de baixíssimo custo computacional. Do lado do servidor, a utilização de *tokens* pseudo-randomicos em todos os formulários para verificação no ato de submissão, garantem satisfatoriamente que a requisição foi feita pelo usuário. Do lado do cliente a proteção pode ser feita por uma mudança nos hábitos de navegação, por exemplo, não navegar por outros sites enquanto possuir uma sessão autenticado em um site que possa ser alvo de ataques (sites de bancos, dentre outros) e sempre encerrar a sessão antes de deixar o site.

Alguns *frameworks* já implementam medidas de prevenção contra os dois tipos de ataques, porém, nem todos[11].

5 Conclusão

O potencial desses dois tipos de ataques é grande, maior ainda quando eles são combinados, e apesar de serem bastante conhecidos, existem muitos desenvolvedores que conhecem apenas superficialmente os mecanismos e a teoria desses ataques, logo não são capazes ou não tem interesse/motivação para projetar/implementar sistemas que tenham alguma proteção contra os mesmos.

O campo do XSS/CSRF é muito amplo e ainda pouco explorado, é freqüente hackers surpreenderem a comunidade de tecnologia da informação com um novo método de ataque. O estudo e conhecimento sobre os ataques por aqueles que estão envolvidos com sistemas Web é imprescindível para que os mesmos se tornem obsoletos ou ineficazes contra sistemas preparados[4].

Referências

[1] Robert Auger. The cross-site request forgery (csrf/xsrf) faq. http://www.cgisecurity.com/csrf-faq.html, April 2008.

[2] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88, New York, NY, USA, 2008. ACM.

[3] CERT. Advisory ca-2000-02 malicious html tags embedded in client web requests. http://www.cert.org/advisories/CA-2000-02.html, February 2000.

[4] Seth Fogie, Jeremiah Grossman, Robert Hansen, Anton Rager, and Petko D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing, 2007.

[5] Apache Software Foundation. Cross site scripting info, February 2000.

[6] James Fulton. Clientsidetrojan. http://www.zope.org/Members/jim/ZopeSecurity/ClientSideTrojan, 2000.

[7] Norman Hardy. The confused deputy (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22:36–38, 1994.

[8] Gary McGraw. Silver bullet talks with jeremiah grossman. *IEEE Security and Privacy*, 7(2):10–14, 2009.

[9] David Ross, Ivan Brugiolo, John Coates, and Michael Roe. Cross-site scripting overview, February 2000.

[10] Peter Watkins. Cross-site request forgeries (csrf, pronounced "sea surf"). http://www.tux.org/ peterw/csrf.txt, June 2001.

[11] William Zeller and Edward W. Felten. Cross-site request forgeries: Exploitation and prevention. http://citp.princeton.edu/csrf/, October 2008.