Tor vs FBI: A história, mitigações e uma análise forense do caso

Guilherme Goldman da Silva[1], Osmar Fernandes Firmino Carpintier[1], Paulo Cabral Sanz[1]

[1]POLI – Universidade Federal do Rio de Janeiro (UFRJ)

{guilhermegoldman,osmar,paulosanz}@poli.ufrj.br

Abstract. *This article aims to analyze aspects of cybersecurity involved during a case of great repercussion involving intrusion practices and data collection from system flaws and vulnerabilities. The case occurred after the FBI invaded anonymous browsing systems in order to obtain the identity of Internet users who used the anonymity provided by the Tor browser to practice illicit actions. In this article, the focus was more on the possibilities of correction/mitigation for related exploits and discuss the effectiveness of these fixes.*

Resumo. *Este artigo possui o intuito de analisar aspectos da cibersegurança envolvidos durante um caso concreto de grande repercussão envolvendo práticas de invasão e coleta de dados a partir de falhas e vulnerabilidades de sistemas. O caso ocorreu a partir de uma invasão do FBI a sistemas de navegação anônima a fim de obter a identidade de internautas que utilizavam da anonimização fornecida pelo browser Tor para praticar ações ilícitas. Nesse trabalho o enfoque foi mais nas possibilidades de correção/mitigação pra exploits dessa natureza e discutir a efetividade dessas correções.*

Introdução

Em 3 de agosto de 2013, o FBI tomou controle da infraestrutura da Freedom Hosting, sistema de nuvem desenvolvido para operar integrado à rede Tor. Fornecendo aos seus clientes acesso à uma máquina virtual acessível através de serviços escondidos, o protocolo da rede Tor esconde o endereço IP dos serviços e portanto permitiu que os donos dos sites de distribuição de pornografia infantil se escondessem ali.

A tomada de controle por parte do FBI permitiu a remoção desses serviços do ar e a análise do conteúdo hospedado na infraestrutura da Freedom Hosting, podendo ser

utilizadas como evidência em processos futuros. O dono da Freedom Hosting foi preso e com o acesso àquela infra-estrutura o FBI pode causar um impacto ainda maior expondo os pedófilos que acessavam os serviços escondidos. Esses usuários também tinham suas identidades protegidas pelo protocolo de cliente do Tor.

Esses serviços PHP eram feitos para usuários que utilizavam navegadores de internet, que compreendem o protocolo HTTP, a renderização de conteúdo HTML5 + CSS3, a decodificação de uma diversidade de tipos de mídia e a execução de código arbitrário de javascript em uma sandbox através de um compilador Just In Time. Bastava encontrar uma vulnerabilidade capaz de ser convertida em RCE em alguma dessas ferramentas, como user-after-free, buffer-overrun, entre outras. Assim seria possível obter o endereço IP de cada uma das pessoas que acessou esses sites ilegais. Não há evidência o suficiente para os prender, mas entrariam para as listas de pessoas suspeitas. E esses dados poderiam ser correlacionados com outras investigações e informações suspeitas para se encontrar e provar crimes.

Para a sorte do FBI, como mostrado por Snowden em seus vazamentos, a NSA tinha ataques capazes de execução remota de código na máquina de usuários do navegador Tor Browser Bundle, que é uma variação do navegador Firefox. Nas versões 10 à 16. O projeto era chamado de EgotisticalGiraffe e era focado na biblioteca de tratamento de XML do Firefox. A versão dessa época, a 17, era mais robusta pois tinha removido tal biblioteca de XML, mas ainda não era claro se a NSA seria capaz de encontrar outros vetores. Encontraram.

No bugzilla, o sistema de rastreamento de problemas do navegador Firefox, continha um ticket criado 4 meses antes do ataque sobre um crash por acesso de memória não mapeada. Algo que costuma ser gerado por um use-after-free, ou um double-free, portanto se mostrava um bug potencial capaz de ser utilizado como primitiva para um RCE.

O usuário que encontrou tal bug reportou que o encontrou através de fuzzing do compilador JIT de Javascript do Firefox, o SpiderMonkey.

Usando esse bug, o use-after-free que causou o crash do bug report, somado a uma sequência de outros bugs que causavam corrupção na heap (buffer overruns), o FBI

foi capaz de quebrar a sandbox do Firefox e realizar uma requisição para serviço do FBI pela clear net, sem anonimização causada pela rede Tor. Contendo o site acessado pelo usuário, o endereço IP, o endereço MAC da placa de rede da máquina e o hostname associado ao Windows. Com isso deanonimizaram os usuários que acessavam os sites que distribuiam pornografia infantil.

Mitigação

Mitigações não servem apenas para os pedófilos expostos com esse ataque, o ataque ficou operacional por um dia, o que significa que atacantes com intenções piores poderiam o adaptar e atacar pessoas não relacionadas com o caso. Além de deanonimizações semelhantes na rede Tor já terem sido utilizadas para perseguir e assassinar militantes e reprimir a população.

A mitigação inicial capaz de proteger o usuário desse exploit e de diversos outros semelhantes seria desabilitar a execução de Javascript, tornando diversas aplicações web 2.0 inutilizáveis, mas que remove uma grande superfície de contato bastante complexa para ataques. É uma recomendação extremamente comum para usuários navegando a rede Tor, sendo inclusive integrada ao navegador do Tor Browser Bundle, através da extensão “NoScript”.

A mitigação individual mais importante é o uso de máquinas virtuais para adicionar uma camada extra de proteção. Por exemplo executando o sistema operacional Tails (The Amnesic Incognito Live System), que garante à nível de kernel que todo acesso à internet é feito pela rede Tor, mas ainda é limitado. Tails já existia na época como um projeto razoavelmente maduro, sendo recomendado na postagem de mitigação que a organização Tor forneceu.

O melhor sistema para proteção seria Whonix, que existia na época, mas ainda era um protótipo que não tinha maturidade para ser muito confiado. Mas hoje já mostra bastante robustez. Whonix é composto de duas máquinas virtuais, a gateway, garantindo proxy pela rede Tor à nível de kernel e a workspace, que realiza todo acesso à rede através da gateway. Mesmo com um RCE no kernel da workspace não é possível evadir o proxy. É necessário quebrar o sandbox da máquina virtual. Algo incrivelmente caro,

visto que toda a infra-estrutura da internet depende delas, então se sua VM for vulnerável provavelmente uma grande parte da internet será também. Do ponto de vista da gestão de recursos não faz sentido descartar um exploit caro assim para atacar indivíduos, é melhor atacar a infraestrutura da internet, tendo acesso irrestrito a diversos serviços de nuvem. Ou é necessário um ataque ao sistema de roteamento das máquinas virtuais que encaminha os pacotes pelo proxy, mas que tem uma superfície de contato tão pequena que se torna impraticável. Esse último também requer injeção de código para ser executado pelo kernel.

Outra mitigação possível, embora não aplicável ainda é a remoção do compilador JIT de Javascript do navegador, o substituindo por um interpretador, que é extremamente menos complexo e não precisa misturar memória de dados com memória de execução. O navegador Edge já faz isso, mas não é possível no Tor Browser Bundle. Além disso, interpretadores são drasticamente mais lentos, mas suas simplicidades permitem uma implementação pouco custosa em linguagens mais seguras, como Rust. Vide o projeto https://github.com/boa-dev/boa.

Se tratando de mitigação real, em menos de 1 dia a comunidade do projeto Firefox forneceu uma correção ao use-after-free utilizado para causar a RCE, que será analisado mais à frente. A análise dessa mitigação se mostra bastante interessante para evitar futuros exploits através de técnicas de programação segura, mesmo quase 10 anos depois do ataque original. Demostra também um aspecto que justifica a narrativa da NSA ter exploits para todas as versões de Firefox recentes, naquele momento. No exploit é possível identificar o uso de um use-after-free e de um buffer-overrun, buffer-overrun no qual não foi corrigido no patch desse bug, possivelmente permanecendo no sistema por muitos anos. Isso significa que uma ferramenta poderosa de converter bugs simples em vulnerabilidades continuou existindo e portanto basta pouco investimento para encontrar outro vetor para tal exploit.

Poderíamos explorar mais como resolver o buffer overrun, algo que a comunidade do Firefox não fez, mas com 10 anos de mudanças no código parecia não agregar tanto. Escolhemos entender o como tornar o firefox estruturalmente mais seguro numa intensidade drasticamente maior que apenas investigar esse problema, que

provavelmente não existe mais. O bug foi detectado originalmente por fuzz testing, produzindo códigos javascript arbitrários e procurando uma sequência que causaria um crash no compilador JIT de javascript do Firefox, visto que crashes costumam ser correlacionados com vulnerabilidades, especialmente os por acesso à memória não mapeada.

Analisando o exploit utilizado pelo FBI, é possível ver que ele é drasticamente diferente e mais críptico do que o crash reportado originalmente, o que nos leva a crer que eles encontraram o mesmo crash também usando fuzzing, mas de forma independente, sem nem saberem que o original tinha sido detectado. Portanto a mitigação concreta do problema é um sistema robusto de fuzzing para engines de Javascript. Algo que se tornou cada vez mais comum nos últimos anos, principalmente com o trabalho do Google Project Zero, que investiga projetos open-source e dedicou muitos recursos ao Chrome, que opera sobre pretextos semelhantes ao Firefox, tendo seus esforços reproduzíveis.

Técnicas de Mitigação Utilizadas

O Project Zero experimentou fuzzing de engines de Javascript de forma geracional, mutação com template e com conhecimento gramatical de Javascript para as variações, enumerando a API e organização semântica do código. Essas variações expuseram uma diversidade de bugs, mas não se mostrou o suficiente para uma eficiência e robustez no fuzzing contínuo. A gramática de Javascript é bastante complexa e usar semânticas inválidas dificilmente causará crashes. Porém as operações de baixo nível que a JIT realiza não são tão complexas assim. Pensando nisso Project Zero criou FuzzIL, uma linguagem intermediária pode ser compilada para Javascript, nessa linguagem então é aplicado o fuzzing de mutação, variando o código e compilando o para Javascript para ser executado. Clonando e modificando os códigos válidos até encontrar crashes, que serão minimizados também através de mutação.

Estratégias de fuzzing tem uma questão fundamental: precisam de muitos recursos alocados constantemente por uma longa duração de tempo para encontrar bugs mais raros, algo muito caro. Isso pode ser acelerado com estratégias de fuzzing melhoradas, mas existe um ponto que se mostra ser mais eficiente adicionar mais CPUs

por mais tempo no problema, do que truques de fuzzing supostamente mais inteligentes, visto que muitos tornarão o processo menos eficiente no final das contas. E só dá pra saber o resultado depois de alocar anos de CPUs/hora para validar a solução.

Com uma evolução da indústria web 2.0 que depende fundamentalmente de uma diversidade de códigos open source que operam com pouco recurso, as maiores empresas começam a alocar recursos para proteger a infraestrutura da internet de forma geral, visto que provavelmente terão algum time interno afetado e seus clientes também serão. E assim nasceram adversários "benevolentes" poderosos o suficiente para competir com recursos de estados nações que têm vastos interesses em vulnerabilidades para viabilizar espionagem.

Além do FuzzIL, existem diversos outros projetos de fuzzing de Javascript bastante interessantes. Como o DIE, projeto do Instituto de Tecnologia da Georgia, que se aproveita de mutações que preservam aspectos. Baseada em preservação de certos aspectos de entradas e saídas de forma estocástica, sem exigir anotações no código. O método de mutação reconhece padrões altamente prováveis de causar recompilação da JIT para explorar bugs como confusão de tipo. As mutações são separadas em mutações de Tipo e de Estrutura. Por exemplo, a estrutura de loop e operações sob o tipo ArrayBuffers são bastante usados para encontrar crashes.

Sendo Javascript uma linguagem turing completa, é possível realizar virtualmente qualquer operação nela, portanto o espaço de fuzzing é infinito. As estratégias então acabam mirando em algumas classes de bugs e utilizando conhecimentos de bugs passados. Por isso, estratégias inovadoras volta e meia são publicadas após encontrar centenas de problemas causados por padrões pouco explorados, em que nenhuma outra estratégia encontraria sem procurar por tal estrutura.

O quão efetivo são essas correções

A execução inicial de FuzzIL, que durou um ano, encontrou 3 vulnerabilidades que poderiam causar RCE em versões publicadas de engines de Javascript com um CVE para cada uma. 2 delas foram no WebKit, 1 foi no SpiderMonkey. Essa foi uma abordagem bastante inovadora, e embora não tenha encontrado um volume grande de

problemas, foram problemas sérios e que não eram encontrados com os métodos tradicionais, mesmo que executados por muito tempo.

Já o motor DIE encontrou 48 bugs entre múltiplos motores de Javascript. Dos quais 16 tem indícios de serem exploitaveis, de onde 12 se tornaram CVEs, gerando um pagamento de 27K USD como bug bounty. Uma questão interessante foi a integração do sistema de logging do ChakraCore, que permitiu encontrar situações que não causavam um crash, mas que demonstravam bugs semânticos na execução do código.

Fuzz testing se mostrou extremamente eficiente ao expor inúmeros problemas na execução de Javascript sendo uma das ferramentas atuais mais importantes na prevenção de ataques. Opera em tempo de execução e é bastante custosa, mas somada à análise estática mais poderosa se mostra chave para garantir segurança da infraestrutura da internet.

Todos os navegadores de internet são basicamente uma máquina virtual para execução de código não confiável de forma eficiente. Garantir que a sandbox não pode ser facilmente quebrada por atacantes com os mais variados níveis de recursos é fundamental para o sucesso da indústria web 2.0 e da 3.0.

Análise do Fix

Tratamos disso no artigo passado, mas vale mencionar mesmo que brevemente como foi realizada a mitigação à nivel de código específica desse ataque. Como mencionado acima o ataque era composto de duas primitivas, um heap spray que causava corrupção de memória, e um use-after-free que executava o binário adicionado em memória evadindo a sandbox e portanto o proxy da rede Tor, expondo o IP dos usuários que acessaram.

O ataque usava o encadeamento de iframes, para explorar um bug na limpeza de memória causada ao recarregar um iframe. Especificamente a callback do evento `readystatechange`, que encadeando o heap spray com iframes que se recarregam de uma forma específica, somada a chamadas ao `window.stop()`. Isso faria com que o código da função callback para o evento fosse dealocado, mas numa situação específica

a callback ainda seria executada. Tradicionalmente causando um acesso à memória não mapeada, que foi encontrado no bug report original.

Preenchendo a heap de forma bastante específica, e se aproveitando de bugs de buffer overrun é possível colocar o código do binário no local da callback, efetivamente realizando uma injeção de código que evade a sandbox e o proxy. O navegador crasha imediatamente e o usuário teve seu IP foi exposto.

A análise do patch elucida bastante sobre o por que certas decisões foram tomadas no código javascript analisado acima.

O primeiro foi o arquivo: content/base/src/nsContentSink.cpp

```
   if (mDocument) {
-    MOZ_ASSERT(mDocument->GetReadyStateEnum() ==
+   MOZ_ASSERT(aTerminated || mDocument->GetReadyStateEnum() ==
                                    nsIDocument::READYSTATE_LOADING, "Bad
readyState");

mDocument->SetReadyStateInternal(nsIDocument::READYSTATE_INTERACTIVE);
    }
```

Esse MOZ_ASSERT não é uma branch que deve ser executada na prática, mas está aí pra evitar justamente que bugs como esse não evoluam para RCEs, causando um crash controlado, limitando-os a apenas DOSs. O fix do bug ocorre em outro arquivo.

O segundo e último arquivo foi docshell/base/nsDocShell.cpp

```
  if (nsIWebNavigation::STOP_CONTENT & aStopFlags) {
      // Stop the document loading
    if (mContentViewer) {
-       mContentViewer->Stop();
+       nsCOMPtr<nsIContentViewer> cv = mContentViewer;
```

```
+       cv->Stop();
    }
  }
```

A mudança é bastante simples, em vez de executar a função `Stop()` numa referência fraca, o smart-pointer é copiado para incrementar a contagem de referências. E só então `Stop()` é chamado. Isso evita que quando `mContentViewer` for a única referência àquele `nslContentViewer` interno, a cópia para `cv` fará com que o contador de referências vá para 2 e portanto `Stop()`" não dealocará variáveis internas que ainda serão acessadas durante a mudança do estado `readystatechange`. Mostrando o porquê do javascript ter chamado "window.stop()" duas vezes com um event listener pra "readystatechange" ativo, no meio de uma sequência de reloads.

Referências

Fuzzing JavaScript Engines with Aspect-preserving Mutation. 2020. https://taesoo.kim/pubs/2020/park:die.pdf

FuzzIL: Coverage Guided Fuzzing for JavaScript Engines. Kit 2018. https://saelo.github.io/papers/thesis.pdf

Tor security advisory: Old Tor Browser Bundles vulnerable. Torblog, 2013. https://blog.torproject.org/tor-security-advisory-old-tor-browser-bundles-vulnerable/. Acesso em: 01 de jun. de 2022.

Attacking Tor: how the NSA targets users' online anonymity. TheGuardian. 2013.https://www.theguardian.com/world/2013/oct/04/tor-attacks-nsa-users-online-anonymity. Acesso em: 01 de jun de 2022.

Crash with onreadystatechange and reload. Bugzilla. 2013. Disponível em:

<https://bugzilla.mozilla.org/show_bug.cgi?id=857883>.Acesso em: 01 de jun de 2022.