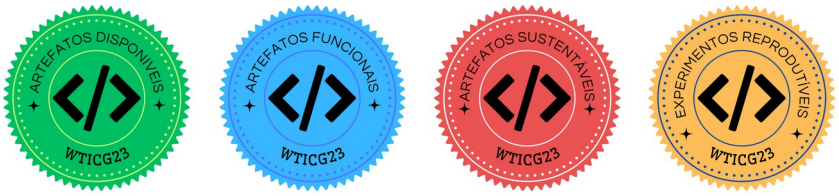

Fuzzing para o Protocolo TLS: Estado da Arte e Comparação de Fuzzers Existentes

Filipe T. Velozo[1], Thiago D. Ferreira[1], Eduardo F. Pacheco[1], Renan C. A. Alves[2], Marcos A. Simplicio Jr.[2], Bruno C. Albertini [2], Daniel M. Batista[1]

[1]Departamento de Ciência da Computação – USP

[2]Departamento de Engenharia de Computação e Sistemas Digitais – USP

{filipetressmann,thiago.duvanel,eduardofp}@usp.br

{renanalves, balbertini, msimplicio}@usp.br, batista@ime.usp.br

Abstract. *Among the various ways to verify the implementation of a protocol, fuzzing tests are worth mentioning, given the good results achieved in recent years both in terms of covering the code that implements a protocol and in terms of finding bugs that can cause security breaches. This paper presents preliminary findings resulting from the investigation of existing fuzzers for the TLS protocol, with the ultimate objective of modifying one of them for testing a new security protocol. It is shown that not all existing fuzzers are functional and that, among those that actually work, the* `tlsfuzzer` *deserves attention as the one to be adapted to verify the implementation of the SPDM protocol.*

Resumo. *Dentre as várias formas de verificar a implementação de um protocolo, os testes fuzzing merecem destaque, dados os bons resultados alcançados nos últimos anos tanto no sentido de cobrir o código que implementa um protocolo, quanto no sentido de encontrar bugs que podem causar falhas de segurança. Este artigo apresenta resultados preliminares decorrentes da investigação de fuzzers existentes para o protocolo TLS, com o objetivo final de modificar um deles para um novo protocolo de segurança. É mostrado que nem todos os fuzzers existentes são funcionais e que, dentre aqueles que de fato funcionam, o* `tlsfuzzer` *merece destaque a ponto de ser considerado como o ideal para ser adaptado para verificar a implementação do protocolo SPDM.*

1. Introdução

O desenvolvimento moderno de software considera que testar o código escrito é algo essencial a ser realizado antes do software ser colocado em produção. No caso de protocolos de comunicação, além de testes locais que verifiquem as implementações dos componentes (funções, métodos, classes, etc...), é importante que toda a máquina de estado definida na especificação do protocolo seja testada, preferencialmente com troca de mensagens entre as partes necessárias para o seu funcionamento. Isso pode ser realizado tanto de forma manual, com desenvolvedores analisando o código em busca de *bugs*, quanto de forma automatizada, com a utilização de software escrito especificamente para testar a implementação do protocolo. No caso de um protocolo cliente-servidor, os testes do servidor podem ser realizados por meio de um cliente desenvolvido especialmente com o

objetivo de testar como o servidor se comporta com diferentes sequências de mensagens, incluindo sequências não esperadas e pacotes mal formados (Em casos onde o código-fonte não esteja disponível, essa é a opção ideal).

Nesse caso da utilização de um cliente "especial" para testar a implementação de um servidor, a técnica fuzzing pode ser empregada. O cliente nesse caso é chamado de *fuzzer* e o servidor sendo avaliado é o SUT (*System Under Test* – Sistema em Teste). Do ponto de vista de segurança, esses testes realizados por *fuzzers* podem expôr vulnerabilidades que vão desde escalada de privilégios [Li et al. 2021] até negação de serviço [Rodriguez and Batista 2023].

Focando em protocolos de segurança, sem dúvida o protocolo TLS (*Transport Layer Security*) merece destaque por ser largamente utilizado na Internet, fornecendo privacidade, integridade e autenticidade na comunicação via rede. Recentemente, o TLS influenciou a especificação do SPDM (*Security Protocol and Data Model*), um padrão aberto proposto pela DMTF (*Distributed Management Task Force*) que define um conjunto de mecanismos e formatos para autenticação de hardware e firmware [DMTF 2023]. O SPDM pode ser usado por exemplo para aumentar a segurança nas leituras e escritas de um disco rígido [Alves et al. 2022]. Dada a relação entre o TLS e o SPDM e a descoberta de falhas em implementações do TLS com o uso de *fuzzers* [Beurdouche et al. 2017], é razoável considerar a seguinte pergunta de pesquisa: **Qual a eficácia de adaptar *fuzzers* do TLS para o SPDM?**

Este artigo apresenta resultados preliminares de um projeto em andamento que visa construir uma suíte de testes automatizados baseada na adaptação de um *fuzzer* do TLS para o SPDM. Ao término do projeto pretende-se comparar a eficácia e a eficiência de tal suíte em relação ao processo manual de testes. Um passo inicial para a construção dessa suíte é a escolha do *fuzzer* TLS e tal escolha é a principal contribuição deste artigo. Oito *fuzzers* para o TLS foram avaliados e, dentre estes, o `tlsfuzzer` foi selecionado para ser modificado para testar o *Responder*, que é a entidade do SPDM equivalente a um servidor na comunicação cliente-servidor. Impressões iniciais da adaptação do `tlsfuzzer` para testar o primeiro par de mensagens do SPDM também são apresentadas neste artigo. Vale ressaltar que todo o software escrito para avaliar os *fuzzers* e a versão preliminar do `tlsfuzzer` modificado estão disponíveis como software livre.

As demais seções deste artigo estão organizadas da seguinte forma: A Seção 2 descreve a metodologia seguida. A Seção 3 apresenta a comparação entre os *fuzzers* TLS e as primeiras mudanças do `tlsfuzzer` para testar a mensagem `GET_VERSION` do SPDM. A Seção 4 conclui o artigo.

2. Metodologia

O primeiro passo realizado consiste na busca pelos *fuzzers* existentes e na seleção daquele que será adaptado para o SPDM. A metodologia seguida nesse passo está apresentada na Subseção 2.1. O segundo passo realizado diz respeito à adaptação do *fuzzer* selecionado para o SPDM. A metodologia seguida nesse passo está apresentada na Subseção 2.2.

2.1. Busca pelos *fuzzers* TLS

Uma busca com a *string* `fuzzer TLS` no Google, no Google Scholar e no GitHub retornou oito *fuzzers* com código-fonte disponível: `TLS-Attacker`, `Cryptofuzz`,

`tlsfuzzer`, `tlspuffin`, `tlsbunny`, `tls-diff-testing`, `NEZHA`, e `FlexTLS`. De acordo com as informações disponíveis nas páginas desses *fuzzers*, muitos deles foram capazes de encontrar *bugs* em implementações do TLS, sendo que alguns desses *bugs* levavam a vulnerabilidades de segurança. Por exemplo, de acordo com as informações disponíveis no repositório do `Cryptofuzz`, este *fuzzer* já encontrou 187 *bugs* em diferentes implementações do TLS, incluindo algumas muito utilizadas como o OpenSSL e o Mbed TLS.

Com a lista de *fuzzers* definida, um conjunto de passos foi realizado a fim de escolher o *fuzzer* ideal a ser adaptado para o SPDM. Esses passos envolveram a compilação e a execução do *fuzzer* para que fosse confirmada a sua capacidade de testar implementações do TLS. Durante as execuções foi feito o monitoramento do tempo de execução, do consumo de CPU e dos pacotes que trafegaram na rede ao testar o servidor TLS disponível com a implementação do OpenSSL versão 3.0.2.

Além das funcionalidades de cada *fuzzer*, também foram analisados a linguagem de programação, a qualidade da documentação, tanto para compilação quanto para execução, e a quantidade de *bugs* encontrados de acordo com as informações públicas fornecidas pelos autores de cada *fuzzer*.

2.2. Adaptação do fuzzer TLS para o SPDM

Uma vez escolhido o *fuzzer* a ser adaptado, o passo seguinte consiste na adaptação do mesmo para o protocolo SPDM. Embora o SPDM tenha se inspirado no TLS, esses dois protocolos são diferentes. A Figura 1 descreve a troca de mensagens iniciais do protocolo de acordo com a versão 1.1.0 da especificação. As duas pontas da comunicação no SPDM são chamadas de *Requester* e *Responder* e são equivalentes a um cliente e a um servidor respectivamente (Em um cenário onde um disco rígido tenha suporte ao SPDM, o *Requester* seria o kernel do sistema operacional e o *Responder* seria o disco propriamente dito).

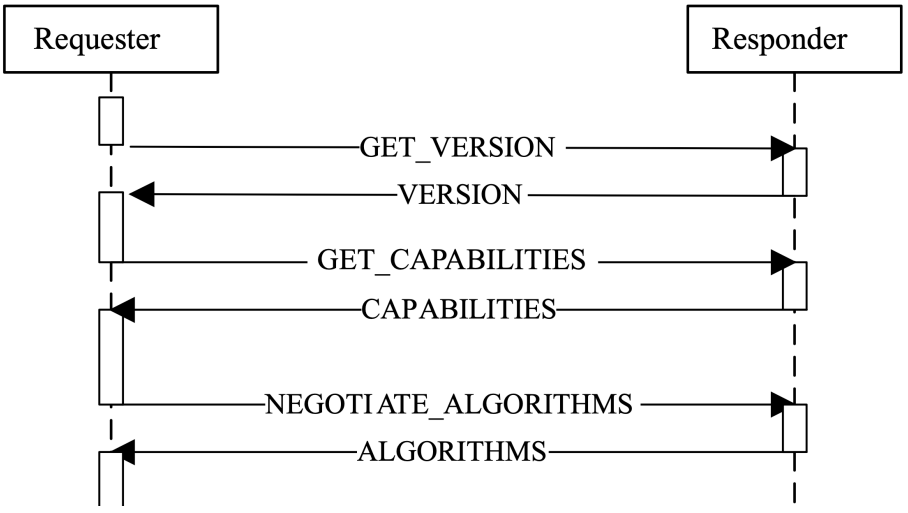

Figura 1. Diagrama de fluxo em alto nível das requisições e respostas iniciais do protocolo SPDM de acordo com [DMTF 2020] (Extraído de [DMTF 2020])

As mensagens `GET_VERSION` e `VERSION` servem para consultar e informar as versões do SPDM suportadas. As mensagens `GET_CAPABILITIES` e `CAPABILITIES`

servem para consultar e informar as funcionalidades suportadas. A lista de funcionalidades é importante para que o *Requester* saiba quais próximas mensagens poderão ser trocadas após as mensagens iniciais. Exemplos de funcionalidades são capacidade de criptografar mensagens, capacidade de manter uma sessão ativa com mensagens de *heartbeat*, capacidade de autenticação mútua, entre outras. As mensagens `NEGOTIATE_ALGORITHMS` e `ALGORITHMS` servem para negociação dos algoritmos criptográficos a serem usados na comunicação segura. Considerando as mensagens iniciais trocadas no protocolo TLS, esses conjunto de três pares de mensagens do SPDM seria equivalente às mensagens `ClientHello` e `ServerHello` do *handshake* do TLS. Logo, ajustes nos testes relacionados a essas duas mensagens em um *fuzzer* para o TLS são necessárias para testar as mensagens iniciais do SPDM.

Embora o SPDM tenha sido projetado para autenticação de hardware e firmware, o que significa que ele é pensado para funcionar sobre barramentos físicos como o PCI Express, ele pode ser implementado como um protocolo de camada de aplicação da Internet usando os serviços providos por protocolos da camada de transporte. Tal implementação facilita os testes do protocolo e isso foi considerado pelo DMTF ao disponibilizar uma biblioteca que implementa o SPDM chamada de OpenSPDM [DMTF 2021]. O código disponível em [DMTF 2021] fornece uma implementação de um *Requester* e uma implementação de um *Responder* úteis para testes. No nosso caso, a versão da OpenSPDM vigente em 15 de Dezembro de 2020 (a última desta data) está sendo considerada como a versão a ser testada pelo *fuzzer*. A escolha por esta versão é pelo fato de que desde esta data uma equipe de desenvolvedores realizou testes manuais na implementação e encontrou alguns *bugs*. Ao usarmos a mesma versão, a comparação entre o *fuzzer* e a verificação manual torna-se justa. No momento, o objetivo é adaptar o *fuzzer* TLS para ele agir como um *Requester*, com o SUT sendo a implementação do *Responder* disponibilizado no OpenSPDM.

3. Resultados Preliminares

Para facilitar a reprodução dos experimentos que avaliaram o desempenho dos *fuzzers* TLS, uma imagem Docker foi criada com um conjunto de *scripts* que faz o download dos *fuzzers*, os compila, inicializa o servidor TLS disponibilizado com o OpenSSL e testa esse servidor com cada *fuzzer*. Cada *fuzzer* é executado 30 vezes e são calculados média e desvio padrão do tempo de execução e do consumo de CPU. Em paralelo à cada execução, os pacotes trocados entre o *fuzzer* e o SUT são capturados para que as mensagens TLS sejam analisadas. Os experimentos foram realizados em um computador com uma CPU AMD RYZEN 7 5800H e 16GB de RAM rodando Arch Linux no Windows 10 x86_64 via WSL. Todos os *scripts* e as instruções para a execução dos experimentos estão disponíveis como software livre no repositório em `https://github.com/th-duvanel/fuzzer-tests/tree/wticg-sbseg2023`. A Tabela 1 apresenta os resultados obtidos juntamente com algumas informações de cada *fuzzer*.

Para cada *fuzzer* foi considerada a versão mais recente disponível nos respectivos repositórios. Com relação aos *bugs* informados nas páginas dos *fuzzers*, não foi possível encontrar essa informação em todos os casos (Para esses *fuzzers*, a posição na tabela aparece com um traço). É possível observar que dos oito *fuzzers*, quatro foram capazes de encontrar *bugs* em implementações do TLS, com destaque para o `Cryptofuzz` e para o

Fuzzer	Data da versão	Linguagem	*bugs* encontrados	Tempo (ms)	CPU (%)
TLS-Attacker	07/07/2023	Java	8	1529,61 +/- 19,25	21,40 +/- 8,31
Cryptofuzz	02/07/2023	C++, C, rust, GO, python e JS	187	–	–
tlsfuzzer	02/07/2023	Python	84	307,04 +/- 0,49	1,57 +/- 0,81
tlspuffin	09/06/2023	Rust, Perl, Python e C	–	–	–
tlsbunny	12/01/2022	Java	3	–	–
tls-diff-testing	13/10/2017	C++, Python e Shell	–	24873,86 +/- 59,85	7,62 +/- 1,80
NEZHA	03/08/2017	C++, C e Shell	–	–	–
FlexTLS	24/07/2015	Python	–	–	–

Tabela 1. Comparação dos *fuzzers* TLS.

tlsfuzzer que encontraram 187 e 84 *bugs* respectivamente. Com relação à execução dos *fuzzers*, dos oito encontrados, apenas três puderam ser executados: TLS-Attacker, tlsfuzzer e tls-diff-testing. Os motivos de não ter sido possível executar os outros cinco *fuzzers* foram principalmente a baixa qualidade da documentação sobre como compilá-los (Cryptofuzz e NEZHA) e sobre como executá-los (tlspuffin, tlsbunny e FlexTLS).

Considerando os três *fuzzers* que puderam ser executados, o que executou mais rápido e consumindo menos recursos foi o tlsfuzzer. Embora o TLS-Attacker e o tls-diff-testing consumam mais recursos e rodem por mais tempo, de acordo com os próprios desenvolvedores eles não foram capazes de encontrar mais *bugs* em implementações de um servidor TLS do que o tlsfuzzer. Analisando os pacotes enviados pelos *fuzzers* que foram executados, é possível notar que todos os três conseguiram testar a mensagem ClientHello, sendo que o tls-diff-testing também foi capaz de enviar pacotes mal-formados, de causar erros de decodificação, de causar erros de parâmetros ilegais e de causar erros de mudança inapropriada de versão do protocolo. Já o tlsfuzzer foi capaz de causar falhas no *handshake* do protocolo.

O TLS-Attacker também foi capaz de enviar mensagens ClientKeyExchange e ChangeCipherSpec, testando assim boa parte do procedimento de *handshake* do protocolo. Entretanto, não foi possível observar pacotes modificados de forma proposital para causar situações inesperadas no servidor, algo importante em um *fuzzer*, fazendo com que o TLS-Attacker fosse descartado. Dentre os dois *fuzzers* restantes, embora o tls-diff-testing gere mais situações inesperadas do que o tlsfuzzer, o fato dele não possuir uma lista de *bugs* encontrados aliado com a ótima documentação das bibliotecas do tlsfuzzer fez com que optássemos por selecionar o tlsfuzzer como o *fuzzer* a ser modificado.

A adaptação do tlsfuzzer para o SPDM encontra-se em andamento. O procedimento que vem sendo seguido tem sido a leitura da versão 1.1.0 da especificação do SPDM e o estudo dos pacotes trocados pelas implementações do *Requester* e do *Responder* disponíveis no código da OpenSPDM. A ideia é implementar os testes seguindo a ordem da troca de mensagens (Figura 1), iniciando portanto pela mensagem GET_VERSION. O código getVersionFuzzer.py disponível no repositório em https://github.com/th-duvanel/fuzzer-tests/tree/wticg-sbseg2023 possui a versão atual do *fuzzer*. O processo de conexão com o ClientHello do TLS foi modificado para o envio da mensagem GET_VERSION do SPDM e, de forma similar, a interpretação da mensagem ServerHello do TLS foi

modificada para suportar a mensagem `VERSION` do SPDM. Como o `tlsfuzzer` utiliza algumas funcionalidades da biblioteca `tlslite`, foi necessário criar o pacote em formato bruto utilizando o `RawSocketWriteGenerator` do `tlsfuzzer`. Inclusive isso é algo que terá que ser cuidadosamente avaliado na continuação da escrita do *fuzzer* (Se muito do que é feito pelo `tlsfuzzer` depender da `tlslite` pode não compensar prosseguir com a sua adaptação para o SPDM).

4. Conclusão e Próximos Passos

Este artigo apresentou a comparação de *fuzzers* para o TLS com o objetivo de selecionar algum deles para ser adaptado para o SPDM. Como resultado da comparação, é proposto que o `tlsfuzzer` seja esse *fuzzer*. No momento ele está sendo modificado para testar a primeira mensagem do SPDM (`GET_VERSION`). Pacotes com essa mensagem já conseguem ser enviados para uma implementação do *Responder* do SPDM e a interpretação das respostas encontra-se em andamento. Também consideraremos reproduzir as vulnerabilidades do TLS encontradas pelos *fuzzers* que fomos capazes de executar.

Agradecimentos

Esta pesquisa é parte do INCT da Internet do Futuro para Cidades Inteligentes, financiado por CNPq (proc. 465446/2014-0), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001 e FAPESP (procs. 14/50937-1 e 15/24485-9). Também é parte do projeto FAPESP proc. 21/06995-0.

Referências

[Alves et al. 2022] Alves, R. C. A., Albertini, B. C., and Simplicio, M. A. (2022). Securing Hard Drives with the Security Protocol and Data Model (SPDM). In *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 446–447.

[Beurdouche et al. 2017] Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.-Y., and Zinzindohoue, J. K. (2017). A Messy State of the Union: Taming the Composite State Machines of TLS. *Commun. ACM*, 60(2):99–107.

[DMTF 2020] DMTF (2020). Security Protocol and Data Model Specification (SPDM) . `https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.1.0.pdf`. Acessado em 17 de Julho de 2023.

[DMTF 2021] DMTF (2021). This openspdm is a sample implementation for the DMTF SPDM specification. `https://github.com/jyao1/openspdm`. Acessado em 17 de Julho de 2023.

[DMTF 2023] DMTF (2023). Security Protocols and Data Models Working Group. `https://www.dmtf.org/standards/spdm`. Acessado em 17 de Julho de 2023.

[Li et al. 2021] Li, R., Diao, W., Li, Z., Du, J., and Guo, S. (2021). Android Custom Permissions Demystified: From Privilege Escalation to Design Shortcomings. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 70–86.

[Rodriguez and Batista 2023] Rodriguez, L. G. A. and Batista, D. M. (2023). Resource-Intensive Fuzzing for MQTT Brokers: State of the Art, Performance Evaluation, and Open Issues. *IEEE Networking Letters*, 5(2):100–104.