Aplicação da Técnica Fuzzing em Testes da Implementação de Referência do SPDM

Thiago D. Ferreira[1], **Renan C. A. Alves**[3],
Bruno C. Albertini[2], **Marcos A. Simplicio Jr.**[2], **Daniel M. Batista**[1]

[1]Depto. Ciência da Computação, Universidade de São Paulo

[2]Depto. Engenharia de Computação e Sistemas Digitais, Universidade de São Paulo

[3]Escola de Artes, Ciências e Humanidades, Universidade de São Paulo

{thiago.duvanel, renanalves, balbertini, msimplicio}@usp.br

batista@ime.usp.br

Abstract. *Automated tests performed during software development are capable of finding flaws early, preventing vulnerabilities ranging from denial of service to privilege escalation. In particular, these automated tests can be performed using the fuzzing technique, which coordinates the sending of unexpected inputs to the software under test. This paper presents preliminary results of* `spdmfuzzer`*, a fuzzer developed to test the reference implementation of the Security Protocols and Data Models (SPDM), a protocol that enables hardware and firmware attestation. In its current publicly available version,* `spdmfuzzer` *has already been able to find unexpected behaviors in the protocol implementation.*

Resumo. *Testes automatizados realizados durante o desenvolvimento de software podem encontrar falhas antecipadamente, evitando vulnerabilidades que vão desde negação de serviço até escalada de privilégios. Em particular, esses testes automatizados podem ser realizados usando a técnica fuzzing, que coordena o envio de entradas inesperadas para o software sendo testado. Este artigo apresenta os resultados preliminares do* `spdmfuzzer`*, um fuzzer que vem sendo desenvolvido para testar a implementação de referência do Security Protocols and Data Models (SPDM), um protocolo voltado para atestação de hardware e firmware. Na sua versão atual disponível publicamente, o* `spdmfuzzer` *já foi capaz de encontrar comportamentos inesperados na implementação.*

1. Introdução

O desenvolvimento de um sistema de software possui diversas fases. Dentre elas, a fase de testes se destaca por possibilitar a descoberta de comportamentos inesperados ou vulnerabilidades de forma antecipada, evitando problemas após a implantação do sistema desenvolvido. Testes podem ser realizados manualmente ou de forma automatizada. No caso de implementações de protocolos de rede, a automação dos testes pode ser realizada com a criação de clientes ou servidores especializados. Nesse caso, a técnica *fuzzing* pode ser empregada para verificar a implementação do código alvo, por meio do envio de entradas inesperadas. O lado que executa os testes é chamado de *fuzzer*, e o lado sendo avaliado é o Sistema sob Teste (*System Under Test* – SUT). Do ponto de vista de segurança,

testes realizados por *fuzzers* podem expor vulnerabilidades que vão desde negação de serviço [Rodriguez and Batista 2023] até escalada de privilégios [Li et al. 2021].

Utilizar protocolos e implementações consolidados, ou inspirar-se neles em vez de criar soluções próprias, é uma boa prática a seguir pois eles já foram analisados e postos a prova. Por exemplo, o TLS (*Transport Layer Security*), largamente usado, inspirou o SPDM (*Security Protocols and Data Models*), um padrão aberto proposto pela DMTF (*Distributed Management Task Force*) para atestação de hardware e firmware [DMTF 2024b]. Por ser relativamente recente, a popularização do SPDM depende de vários fatores, entre eles a atestação de sua efetiva segurança. Ações nesse sentido foram realizadas nos últimos anos por vários grupos envolvidos na especificação do protocolo e no desenvolvimento da `libSPDM` [DMTF 2024a] (anteriormente chamada de `openSPDM` [DMTF 2021]), uma implementação de referência do SPDM. Essas ações foram manuais ou automatizadas. No caso automatizado, testes fuzzing foram empregados com a utilização de fuzzers genéricos como o `AFL` (*American Fuzzy Lop*) [AFL 2021] e o `OSS_Fuzz` [OSS-Fuzz 2023]. Apesar desses testes que vêm sendo realizados nos últimos anos, a seguinte questão de pesquisa ainda não foi respondida: **Qual o desempenho, em termos de capacidade de encontrar falhas, de um *fuzzer* escrito especificamente para o SPDM tirando proveito do conhecimento da gramática do protocolo?** Este artigo descreve os resultados preliminares de um projeto de iniciação científica que visa responder essa pergunta por meio de um *fuzzer* específico para o SPDM, chamado de `spdmfuzzer`. Até o momento, o `spdmfuzzer` é capaz de testar o primeiro par de mensagens do protocolo e já foi capaz de identificar comportamentos que, ao menos em princípio, violam a especificação do SPDM.

O restante deste artigo está organizado como segue. A Seção 2 resume trabalhos relacionados. A Seção 3 descreve o método usado no desenvolvimento do `spdmfuzzer`, sua arquitetura, campanha e forma de aplicação. A Seção 4 apresenta os resultados alcançados até o momento. A Seção 5 conclui o artigo e lista os próximos passos.

2. Trabalhos relacionados

Em [Cremers et al. 2023], é realizada uma análise formal da versão 1.2 do SPDM, com a descrição de sua máquina de estados, que salienta a importância de uma abordagem linear na troca de mensagens. O `spdmfuzzer` complementa o trabalho em [Cremers et al. 2023] pelo fato de testar o protocolo em uma implementação real.

O uso do SPDM na prática pode ser visto em [Alves et al. 2022], onde foi feita uma análise e experimentação do protocolo para autenticação de dispositivos de armazenamento. O trabalho descreve o processo de autenticação e qual a importância de cada uma das mensagens, o que contribuiu para o projeto do `spdmfuzzer`.

Existem vários métodos para a geração das mensagens em um *fuzzer*, como mutação, que utiliza mensagens pré montadas para alteração aleatória de seus bytes, ou gramática, que utiliza a estrutura das mensagens para formar dados semi aleatórios. Na comparação entre ambas abordagens em [Rodriguez and Batista 2021], conclui-se que *fuzzers* de gramática possuem boa eficiência e cobertura de código, principalmente quando aplicados em protocolos de rede. Por isso, o `spdmfuzer` é um *fuzzer* de gramática.

Por ser um protocolo novo, existem poucas ferramentas disponíveis para facili-

tar a depuração de implementações do SPDM. Por causa disso, durante o desenvolvimento, o dissecador SPDM-WiD [Ferreira et al. 2024] vem sendo usado, permitindo a interpretação dos pacotes do protocolo no `Wireshark`.

Com base na busca da literatura, podemos afirmar que o `spdmfuzzer` é o primeiro *fuzzer* dedicado para testes automatizados, especificamente, do SPDM.

3. Metodologia

O SPDM é um protocolo *request-response*, ou seja, temos um *Requester*, que busca autenticar e descobrir as versões e algoritmos suportados pelo *Responder*. Além disso, ele é de baixo nível pois sua atuação se dá nos momentos iniciais de inicialização da máquina, em que a troca de mensagens é feita no barramento da placa mãe. Para fins de entendimento, podemos comparar o *Requester* ao *driver* de um dispositivo de hardware, que atua ao lado do *kernel* do sistema operacional, e o *Responder* ao dispositivo propriamente dito, como um disco rígido [Alves et al. 2022], que tenta se provar autêntico por meio de troca de certificados e checagem de capacidades requisitadas por aquele *driver*.

No caso de testes *fuzzing*, poderíamos aplicá-lo tanto como um *Requester* quanto um *Responder*. Nesse trabalho, o *fuzzer* toma o lugar do *Responder*, visando observar inconsistências nas respostas do *driver* a partir da geração de mensagens por gramática.

3.1. Campanha de *fuzzing*

Respostas inesperadas podem aparecer de diversas formas possíveis, e, para explorá-las, o *fuzzer* deve possuir conhecimento tanto das entradas quanto dos estados possíveis do protocolo, para manter consistência de execução durante sua campanha de *fuzzing* [Chen et al. 2019]. Como primeiro passo do desenvolvimento do `spdmfuzzer`, foi decidido explorar o par de mensagens `GET_VERSION` e `VERSION`, que serão o foco deste artigo, por serem responsáveis por iniciar a comunicação (Figura 1).

Figura 1. Mensagens iniciais do SPDM (Adaptado de [DMTF 2020])

No caso, o *Requester* vai enviar um `GET_VERSION` para o *fuzzer*, que vai respondê-lo de forma que possa receber alguma resposta inesperada. Para tirar proveito disso, como o `spdmfuzzer` é baseado em gramática, a formatação dos pacotes deve seguir uma semi aleatorização que faça sentido no protocolo. Ao observar a estrutura de seus pacotes lendo a especificação, percebemos que há um cabeçalho comum a todos, que define o tipo do pacote, e um campo específico, que deve ser explorado (Figura 2).

Apesar do `spdmfuzzer` ter uma campanha flexível, decidida por poucas mudanças em linhas de código, para este artigo, seu comportamento foi definido da seguinte forma: em relação ao tamanho das mensagens, como o pacote possui um campo que influencia seu tamanho (`Quantidade de Versões`), apenas esse valor será aleatorizado pois não há necessidade de aleatorizar seu tamanho e preencher o pacote com

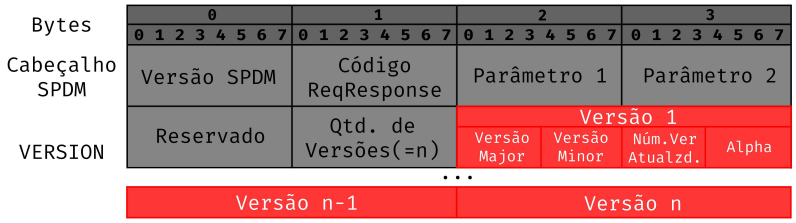

Figura 2. Dissecação do pacote `VERSION`

bytes avulsos. Sobre a ordem de envio, por ser um protocolo extremamente engessado, ou seja, não há mudança de estado se o tipo de pacote recebido, definido pelo campo `Código Request Response`, não for o esperado, a ordem é mantida. Além disso, o comportamento geral do `spdmfuzer` é que, caso ele encontre uma requisição inesperada, ele armazena-a e começa a "fuzzificar" a próxima até conseguir uma autenticação completa. Isso é considerado uma rodada de teste finalizada.

3.2. Aplicação do *fuzzer*

Como o SPDM é de baixo nível, o uso do *fuzzer* torna-se de difícil manutenção, já que seria necessário uma aplicação do protocolo em ambiente físico ou virtual envolvendo a emulação de vários outros componentes. Contudo, por ser da camada de aplicação, seu desenvolvimento foi voltado a ser extremamente portável para suportar diversos protocolos da camada de transporte. Ao utilizá-lo sem alterações, o protocolo MCTP (*Management Component Transport Protocol*) entra em cena, voltado à comunicação de *hardware*. Contudo, em seu próprio repositório [DMTF 2021] é possível encontrar bibliotecas e uma implementação em C de um *Requester* e de um *Responder* utilizando TCP, mas sem descartar os pacotes do MCTP, para fins de estudo. Isso possibilitou o uso do `spdmfuzer` de forma direta, com fácil compilação e instalação.

O repositório foi montado de forma que fosse possível compilar automaticamente tanto a biblioteca `openSPDM`[1] quanto o `spdmfuzzer`. Além disso, ao executá-lo, ele automaticamente vai executar o binário relativo ao *Requester*, já que o *fuzzer* atua como *Responder* nesse caso. Caso ele receba uma mensagem de desconexão após o envio de uma mensagem (que é o comportamento esperado), ele automaticamente libera o *socket* fechado e executa novamente o binário legítimo, reiniciando o processo. Além disso, vale ressaltar que o *fuzzer* também foi pensado para ser modular, ou seja, caso seja necessário aplicá-lo em ambientes emulados que utilizem C, como o incluso no [SPDM-WiD 2024], ou que não utilizem o TCP, é possível, bastando alterar ou adicionar poucas linhas de código, já que ele foi programado orientado a objetos em C++. O código-fonte e toda a instrução de compilação e execução estão disponíveis em `https://github.com/th-duvanel/spdmfuzzer/tree/sbseg24`.

4. Resultados preliminares

Apenas testando a primeira mensagem com o `spdmfuzzer` já foi possível observar um comportamento inesperado (Figura 4). Após colocá-lo em uma simples rodada de testes, utilizando-se da aleatorização por gramática do pacote, ao observar especificamente

[1] Apesar de ser uma implementação antiga, ela foi escolhida por ter sido extensivamente testada manualmente nos últimos anos, servindo de uma boa referência para a comparação da eficácia do `spdmfuzzer`

o comportamento da variável `Qtd. de Versões` de um byte (ou seja, até 255), houve um avanço do estado do protocolo para mensagens com essa variável igual a 2, independente do conteúdo do restante dos bytes que o complementam. No caso do `VERSION`, ele é necessário para o *Requester* decidir uma versão comum que ele e o *Responder* suportem. A versão padrão de suporte é a 0x1, que é a primeira versão. Contudo, mesmo não especificando-a como uma das suportadas, o protocolo ignora esse fato, enviando o próximo pacote de tipo *request*: o `GET_CAPABILITIES` (Figura 3).

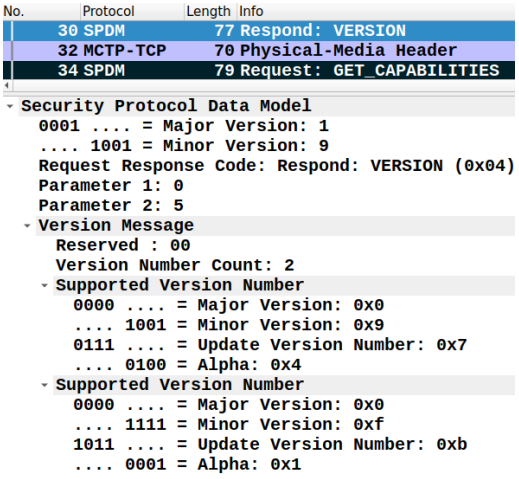

Figura 3. Captura no `SPDM-WiD`

```
# [+] => Received buffer:
        05 10 84 00 00

# [+] => Sent command: 00 00 00 01
# [+] => Sent transport type: 00 00 00 01
# [+] => Sent buffer size: 00 00 00 0b
# [+] => Sent buffer:
        05 19 04 00 05 00 02 09 74 0f b1

# [+] => Received command: 00 00 00 01
# [+] => Received transport type: 00 00 00 01
# [+] => Received buffer size: 00 00 00 05
# [+] => Received buffer:
        05 10 e1 00 00
# [+] => wow! this is not expected.
```

Figura 4. Resposta inesperada

De acordo com a especificação do SPDM [DMTF 2019], se o *Requester* responder a um `VERSION` mal formatado, já temos uma inconsistência, pois a conexão não deve continuar caso *Requester* e *Responder* não possuam versão compatível em comum, que é exatamente o caso observado, já que temos valores de versões que nem existem ainda, principalmente ao considerar que a versão usada para testagem é a primeira. Ou seja, não faz sentido uma implementação da versão 1.0 suportar versões maiores.

5. Conclusão e próximos passos

Protocolos de autenticação, como o SPDM, precisam ser bem implementados para garantir que os nós em comunicação são de fato quem informam ser. Falhas na implementação podem invalidar toda a comunicação posterior. Este artigo apresentou os resultados preliminares do `spdmfuzzer`, um *fuzzer* para o protocolo SPDM. Apesar do pacote `VERSION` possuir uma estrutura mais simples comparada aos pacotes subsequentes,os resultados já foram animadores, pois com apenas um pacote modificado de forma gramatical, foi encontrada uma resposta inesperada, que talvez leve a uma possível vulnerabilidade. No momento, o suporte a outras mensagens está sendo implementado, o que vai permitir a realização de uma rodada completa de testes. Após isso, serão feitas comparações com vulnerabilidades encontradas de forma manual, além da análise de todas as inconsistências encontradas ao longo dos testes.

Agradecimentos

Esta pesquisa é parte do CPE SMARTNESS (FAPESP 21/00199-8). Também é suportada pelos projetos FAPESP 20/09850-0, 21/06995-0, 23/16002-4, 14/50937-1 e 15/24485-9, CNPq 307732/2023-1 e 465446/2014-0 e CAPES 001.

Referências

[AFL 2021] AFL (2021). Fuzzing with afl-fuzz — AFL 2.53b documentation. `https://afl-1.readthedocs.io/en/latest/fuzzing.html`. Acessado em 2 de Julho de 2024.

[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.

[Chen et al. 2019] Chen, Y., lan, T., and Venkataramani, G. (2019). Exploring effective fuzzing strategies to analyze communication protocols. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, FEAST'19, page 17–23, New York, NY, USA. Association for Computing Machinery.

[Cremers et al. 2023] Cremers, C., Dax, A., and Naska, A. (2023). Formal analysis of SPDM: Security protocol and data model version 1.2. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6611–6628, Anaheim, CA. USENIX Association.

[DMTF 2019] DMTF (2019). Security Protocol and Data Model Specification (SPDM) . `https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.0.0.pdf`. Acessado em 1 de Julho de 2024.

[DMTF 2020] DMTF (2020). Security Protocol and Data Model Specification (SPDM). `https://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.1.1.pdf`. Acessado em 3 de Julho de 2024.

[DMTF 2021] DMTF (2021). This openspdm is a sample implementation for the DMTF SPDM specification. `https://github.com/jyao1/openspdm`. Acessado em 1 de Julho de 2024.

[DMTF 2024a] DMTF (2024a). DMTF/libspdm. `https://github.com/DMTF/libspdm`. Acessado em 2 de Julho de 2024.

[DMTF 2024b] DMTF (2024b). Security Protocols and Data Models Working Group. `https://www.dmtf.org/standards/spdm`. Acessado em 2 de Julho de 2024.

[Ferreira et al. 2024] Ferreira, T. D., Freitas, O. F., Alves, R. C. A. A., Simplicio, M. A., Albertini, B. C., and Batista, D. M. (2024). SPDM-WiD: Uma Ferramenta para Inspeção de Pacotes do Security Protocol Data Model (SPDM). In *2024 SBC Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*. `https://smartness2030.tech/wp-content/uploads/2024/05/Camera-Ready-SPDM_Ferramentas_SBRC2024.pdf`. Acessado em 2 de Julho de 2024.

[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.

[OSS-Fuzz 2023] OSS-Fuzz (2023). OSS-Fuzz — Documentation for OSS-Fuzz. `https://google.github.io/oss-fuzz/`. Acessado em 2 de Julho de 2024.

[Rodriguez and Batista 2021] Rodriguez, L. G. and Batista, D. M. (2021). Towards Improving Fuzzer Efficiency for the MQTT Protocol. In *2021 IEEE Symposium on Computers and Communications (ISCC)*, pages 1–7.

[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.

[SPDM-WiD 2024] SPDM-WiD (2024). SPDM (Security Protocol Data Model) dissector and packet exporter for Wireshark. `https://github.com/th-duvanel/spdm-wid`. Acessado em 2 de Julho de 2024.