Volga CTF

PIWAS-driven exploitation

whoami

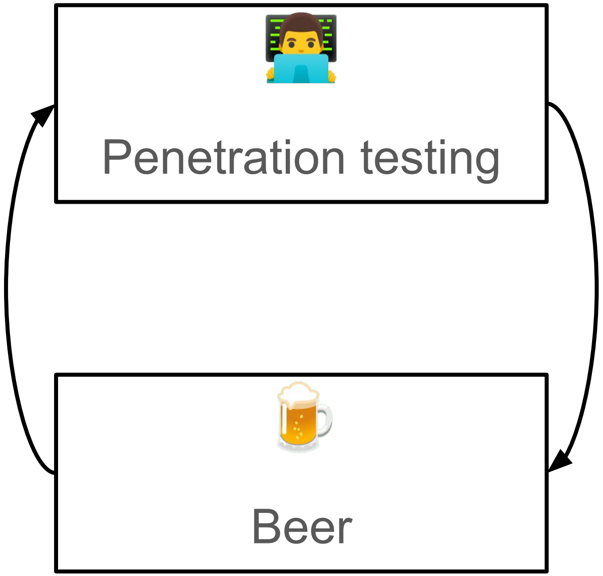
Pavel Toporkov

👨‍💻 Penetration testing at Kaspersky

🍺 Beer lover

Join us and drink beer with me!

Agenda



👨‍💻

Penetration testing

🍺

Beer

🍺 PIWAS

Beer (rus. piwas) is an alcoholic beverage produced by the brewing and fermentation of starches from cereal grains—most commonly malted barley, although wheat, maize (corn), rice, and oats are also used

👨‍💻 Penetration testing

Trends of modern infrastructure:

- 🍻 Microservice architecture
- 🍻 Containers hardening
  - 🍻 No access to the internet
  - 🍻 No excessive capabilities

Comparison table

	Beer	Microservice Architecture
Introduced	11000 BC	2005
Makes people happier at first	✅	✅
Gives a headache afterwards	✅	✅
Looked like a mess at the beginning	✅	✅

Comparison table

	Beer	Microservice Architecture
	11000 BC	2005
at first	✅	✅
wards	✅	
ne	✅	

👨‍💻 Penetration testing

Assuming we got RCE on the application server.

We cannot:

- 🍻 Upload a webshell
- 🍻 Get a reverse shell
- 🍻 Modify running processes (no SYS_PTRACE)

👨‍💻 Penetration testing

Assuming we got RCE on the application server.

We want to:

- 🍻 Be able to inject a web shell to a running application process
- 🍻 Be able to inject a websockets SOCKS proxy to a running application process
- 🍻 Have a cool beer related name

🍺 🍺 🍺

Process

Injected

Webshell

And

Socks

🍺 Piwas

Piwas is a tool to inject your payload (webshell and SOCKS proxy) to a running processes.

It uses internal language/framework mechanisms to avoid using ptrace

Currently following environments are supported:

- 🍻 NodeJS+Express
- 🍻 Java+Tomcat

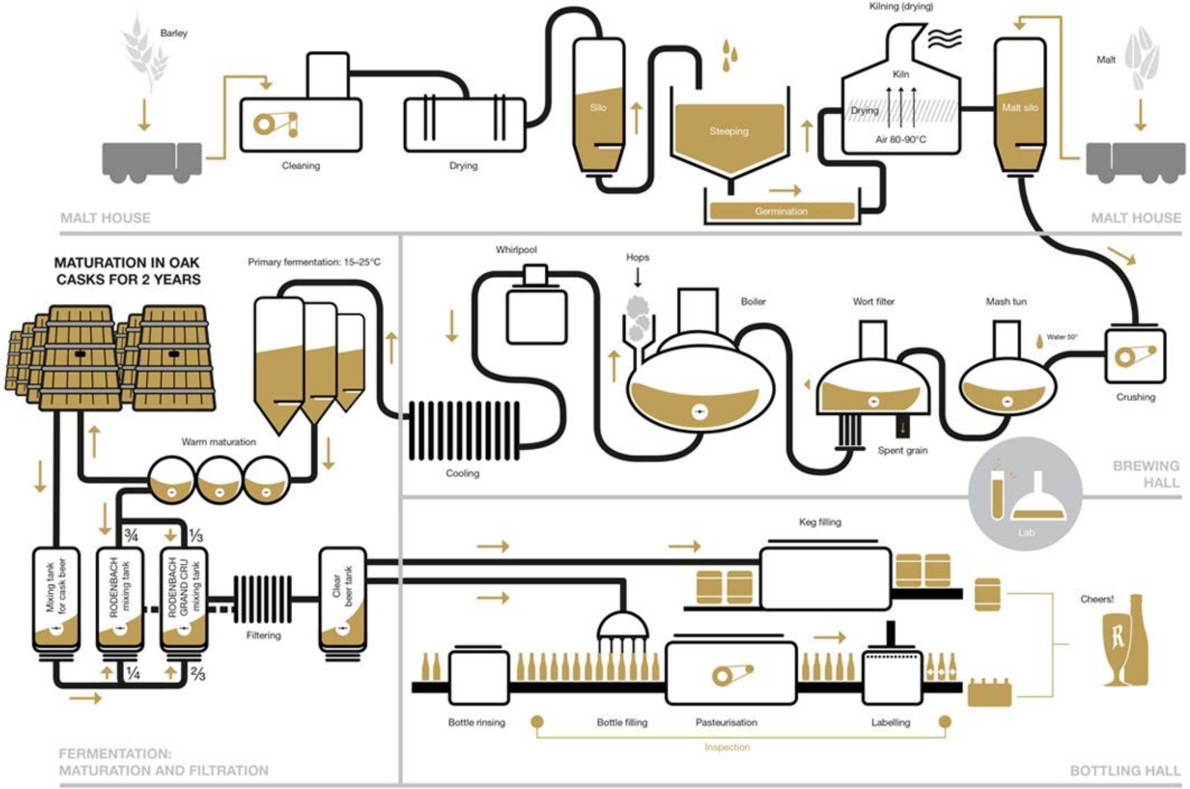
🍺 Piwas

Current implementation registers following endpoints in the running application:

- 🍻 /{SHA1}_sh - OS commands webshell
- 🍻 /{SHA1}_eval - eval webshell
- 🍻 /{SHA1}_socks - SOCKS over WebSockets

where {SHA1} is a SHA1 hashsum of corresponding injecting payload

🍺 Implementation details 🍺

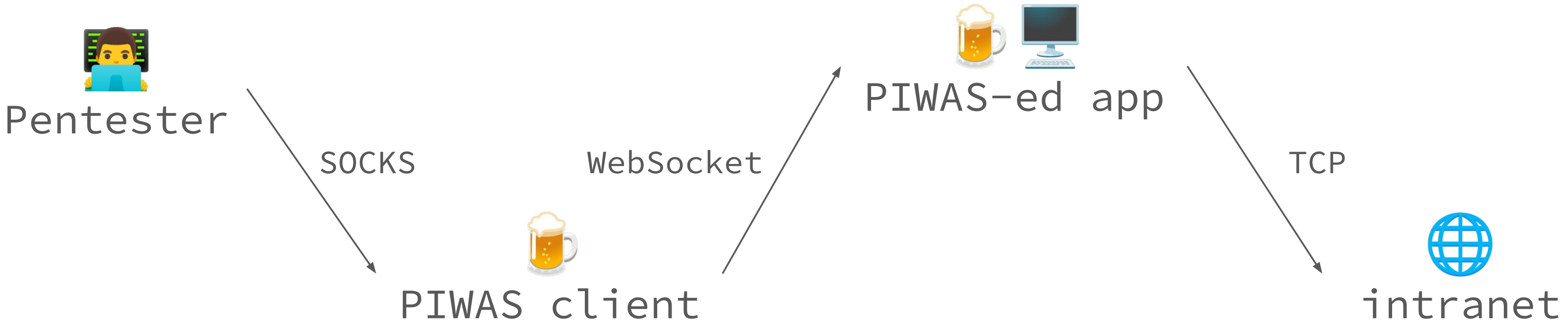


🍺 PIWAS

Current version of PIWAS consist of three parts

- 🍻 NodeJS injector
- 🍻 Java injector
- 🍻 SOCKS proxy client

🧦 + 🍺



Pentester

SOCKS

🍺 PIWAS client

WebSocket

🍺🖥️ PIWAS-ed app

TCP

🌐 intranet

```
$ ./piwas-client -url "ws://host/$(sha1sum piwas.jar|cut -d ' ' -f 1)_socks" &

$ proxychains curl http://internal/
```

NodeJS + 🍺

The loader uses Node.JS debugger protocol.

1. send SIGUSR1 signal to a target process in order to turn the NodeJS debugger on.
2. checks for the new listening ports (or tries all ports listened by the app in case if debugger was already enabled)
3. checks if the port provides a NodeJS debugger API.

https://nodejs.org/en/learn/getting-started/debugging

NodeJS + 🍺

In order to find the Express.js Application instance object, the application changes the the prototype of **Router** class, modifying the **handle** method.

The modified method calls the original method, finds the **Application** and socket server instance objects in the methods context, registers both webshell and websocket socks endpoints and returns the **Router.handle** method to the original state.

Note that we need to make at least one request in order to make all the endpoints being registered

Java + 🍺

PIWAS uses Java Attach API to inject the payload into a running Java process.

Current PIWAS version requires Java 8+, and tested on tomcat based environments (SpringBoot will also work).

Java + 🍺

The injected payload tries to find **javax.servlet.http.HttpServlet** (or **jakarta.servlet.http.HttpServlet** for the newer versions of environment) class in the memory and then tries to modify the service method, adding the PIWAS code to the beginning of the method. This code will be executed on every HTTP request to the application and basically makes two things:

- It checks if we already have registered an websocket endpoint. Otherwise it registers an websocket endpoint which will provide a SOCKS tunnel. **Note that we need to make at least one request in order to make websocket endpoint being registered**
- Checks if the request path contains {SHA1}_sh/{SHA1}_eval strings and reroutes execution flow to the webshell code.

🍺 + 🧃

🍺 + 🧃

Previous approach requires OS commands to be executed

In some cases we can avoid it, being as stealthy as possible (e.g. when we are originally in the eval execution context)

🍺 + 🧃

```
# NodeJS PIWAS injecting
$ node piwas.js $(pidof node) # (IoC!)

# Java PIWAS injecting
$ java -jar piwas.jar $(pidof java) # (IoC!)
```

🍺 + 🧃

The idea is to use Java Attach API to inject the payload. The Java attach API protocol is following:

- Loader process creates the .attach$(pid) empty file in the target process working dir (IoC!)
- Loader process sends SIGQUIT signal to the target process
- Target process will create /tmp/.java_pid$(pid) UNIX socket file (IoC!)
- Loader process sends a load command through the UNIX socket
- Target process will load arbitrary JAR or SO file from the filesystem (IoC!)

https://docs.oracle.com/javase/8/docs/technotes/guides/attach/
https://github.com/paul-axe/slides/blob/master/spbctf%202021%20-%20JVMyacni%20Stories.pdf

🍺 + 🧃

NodeJS injection script is made to be able running in both OS command and eval execution contexts.

```
eval(global.__injected = true;{PIWAS_CONTENT})
```

Java eval-based injection TBD

TODO

- 🍻 Drink beer
- 🍻 Add eval-based injection in Java processes
- 🍻 Support more frameworks/languages

questions?