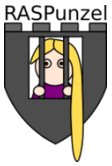
RASPunzel: A Novel RASP Solution



What is RASP?

Runtime Application Self-Protection Runtime (RASP) is a security mechanism that is built into applications to protect themselves from attacks. RASP monitors the attack vectors of an application and detects security events.

The team that manages the application can identify what are security threats they want the RASP solution to handle. This is achieved without requiring the developer to modify their code to accommodate a RASP tool. The security monitors are woven into the application at build time or at runtime. Because no changes to the application development life cycle is required, RASP solutions are relatively inexpensive to deploy.

There are various ways of deploying RASP solutions. In monitoring mode, a RASP tool will only report the security event. The application team can then decide how to handle the reported event. In full protection mode a RASP tool will prevent this security event from occurring. This can be achieved in many ways including stopping the execution of the application, terminating the user's session or raising an exception for the current request. RASP solutions are configurable, and the application can identify which events need to be reported and which events need to be blocked.

RASP solutions, unlike Web Application Firewalls (WAF) provide an application-aware level of protection. A RASP solution can also analyse the severity of a security event by evaluating its impact on the application. This could include change in the application's configuration and exfiltration of sensitive data. WAF-based solutions do not have this level of visibility of the application and can only report on incoming requests. This makes RASP-based solutions more accurate and can reduce both false positives and false negatives. NIST's SP 800-53B recommends a state-of-the-art RASP solution.

How Does RASPunzel Work?

RASPunzel, unlike other RASP solutions, uses an **allowlist** approach to provide security. That is, the security posture is about what is permitted rather than what is disallowed (or the deny-list approach). From a security perspective, using deny-lists is challenging. Every new vulnerability requires the

security operations (or even the developer) to update the deny-list. This results in a whack-a-mole approach to security. Using an allowlist provides a better security posture. Only legitimate behaviours need to be identified. All behaviours not mentioned in the allow-list are reported and flagged. They can also be prohibited. Hence an application protected by RASPunzel is not affected by newly disclosed vulnerabilities unless the vulnerable behaviour happens to be permitted by the allowlist.

RASPunzel uses patented[1] techniques to generate the allowlists **automatically**. RASPunzel has two distinct phases. In the first phase the synthesis engine in RASPunzel is exercised via both permitted and forbidden behaviours. This can be during the testing phase. This means that the application team does not have to do any extra work to improve the security posture.

RASPunzel, generalises the permitted behaviours using the semantics of the defect type being supported. For instance, to prevent SQL injections, RASPunzel uses an information flow semantics to identify the consequences of the SQL query. RASPunzel's information flow model allows fine-grained control of information disclosure. For instance, even users who have access to the entire table may not be able to disclose values from certain columns. At the end of the second phase RASPunzel generates the allowlist. RASPunzel's generalisation ensures that none of the forbidden behaviours are permitted.

RASPunzel's allowist list generation is **context sensitive**. A single security-sensitive operation could be executed on different program paths. These program paths represent different usage contexts. Hence each path to a security-sensitive operation will have its own allowlist. This is explained later via an example.

RASPunzel also adopts a customer centric view. Each deployment of the application can have its own customized security posture. This permits different allowlists based on the configuration and test cases used. Those who manage the deployment of the application can influence the allowlist generation via suitable tests.

RASPunzel has two ways to generate the security monitor based on the generated allowlist. The first way is to weave the monitor at build time using the Graal Native Image[2] system. The second is to weave the monitor at runtime using a Java agent.

[1] Patent applications under review.
[2] https://www.graalvm.org/22.1/reference-manual/native-image/

In both cases, the woven monitor tracks the behaviour of the application. The monitors can be configured to log, alert (e.g., to a SIEM system) or block behaviours that are not permitted by the allowlist.

Simple Example

Consider a table, called `movies`, that has `name`, `director`, `genre`, `release_year`, and `profit` as its columns and the query `'SELECT profit FROM movies WHERE (name = 'Iron Man 2')'` at a particular program point. On one execution path, the following allowlist is generated for the user of the application:

Allowed disclosure: `movies.name, movies.director, movies.genre, movies.release_year, movies.profit`

while on another execution path for another user, the following allowlist is generated:

Allowed disclosure: movies.`name, movies.director, movies.genre, movies.release_year~~, movies.profit~~`

This demonstrates the contextual and fine-grained nature of RASPunzel's information flow-based allowlists.

When the first path is executed, RASPunzel reports

```
Executing trusted query 'SELECT profit FROM movies WHERE (name = 'Iron Man 2')'
```

This is because the disclosure of the column `profit` is *permitted* by the allowlist. However, when the second path is executed, RASPunzel, in alerting mode, reports

ALERT: Executing **untrusted** query `'SELECT profit FROM movies WHERE (name = 'Iron Man 2')'`

The query is invalid on this path because the disclosure of the column `profit` is *not permitted* by the generated allowlist.

Autonomous RASPunzel

Another important aspect of RASPunzel's design is the autonomous nature of updating the allowlists[3]. Because RASPunzel monitors the execution of the application, it can gather real observations. These observations can be used to refine the existing allowlist to obtain a more precise and accurate allowlist. Thus RASPunzel automatically keeps improving the context specific protection it synthesised initially.

The allowlist generation and the monitoring of the application are seamless and require very little or no developer/user interaction. This allows RASPunzel to be deployed in a seamless fashion.

The high-level architecture is shown in Figure 1.

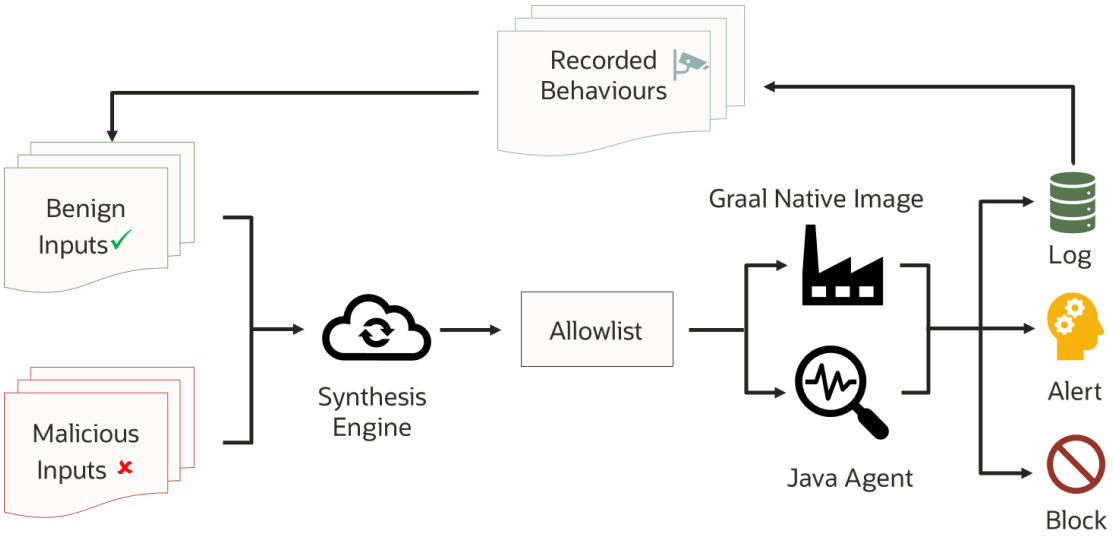


Figure 1: Architecture of RASPunzel

The main benefits of RASPunzel are:

1. Real-time protection: RASPunzel can detect malicious behaviour such as deviant and potentially malicious deserializations (including JNDI-based attacks such as Log4Shell) and SQL injections.
2. RASPunzel has full visibility of the application. Hence its security alerts have full context of deviant behaviour.

[3] This feature will be forthcoming soon.

3. RASPunzel has a fine-grained information flow-based protection mechanism which is context-sensitive and customer centric.
4. RASPunzel can protect against both known and unknown attacks as only permitted behaviour will pass the monitor.
5. RASPunzel can protect against zero-day exploits. If the zero-day exploit is not related to the allowlist, no change is required. The application remains protected against this exploit. If the allowlist is impacted by the exploit, it can be updated to have an upated allowlist. RASPunzel gives time to the development team to patch the application.
6. The RASPunzel monitors are suitable for both standard Java web applications as well as cloud native applications that leverage the Graal Native Image technology.
7. Easily integrable in a DevSecOps system.