TESTABLE: Testability-driven security and privacy testing for Web Applications

Abstract—**Modern web applications play a pivotal role in our digital society. Motivated by the many security vulnerabilities and data breaches routinely reported on those applications, we initiated the EU TESTABLE research project to address the main challenges of building and maintaining web applications secure and privacy-friendly. The ultimate goal is to lay the foundations for a new integration of security and privacy into the software development lifecycle (SDLC), by proposing the novel idea of combining two metrics to quantify the security and privacy risks of a program, i.e., the testability of the codebase (via the novel concept of "testability patterns") and the indicators for vulnerable behaviors.**

We have already achieved promising results by applying our research proposal in the area of static analysis security testing (SAST) [2]. Hundreds of tarpits—code instructions challenging for SAST tools—have been identified and captured in testability patterns that we used to measure SAST tools effectiveness as well as to improve the "SAST testability" of open source applications via code refactoring routines removing the tarpits. More than 180 new vulnerabilities have been uncovered after refactoring and confirmed by open source applications' owners. We believe similar promising results can also be achieved in other areas such as dynamic analysis, privacy, and machine learning.

Index Terms—**web, testability, risk, measurement, metrics, security, privacy, ML**

1. Introduction

Web applications are ubiquitous, and they are used in a multitude of different domains. According to the 2020 Edgescan Security Report, "Web application security is where the majority of risk still resides" [1]. This is confirmed by the fact that most of the recent data breaches took advantage of the poor security of web applications.

Software security testing plays a fundamental role to mitigate this problem. In particular, developers regularly use static code analysis and automated testing tools to verify their application and identify vulnerabilities. But existing solutions are often limited in their ability to automatically discover security problems. When problems are not detected is always challenging to know for sure what the testing tools left uncovered.

While threat modeling and risk assessment methodologies are often adopted by industry at early phases of the software development lifecycle, they are too far from the implementation details and therefore they can only set high-level recommendations for later phases, such as testing. Computing risk measures that are closer to the application code itself can be used to complement early risk assessment phases, and could help to prioritize the testing effort. However, this area has so far mainly focused on determining the likelihood that an application contains a vulnerability (what we call **vulnerability indicators**). Therefore, even when these indicators exist, they provide little guidance to the developers on how the corresponding risk could be mitigated. For instance, knowing that a web application has a high risk of containing vulnerabilities (even when none have been detected during development and testing) is a piece of information that can be difficult to translate into actionable insights.

The poor state of web security is further exacerbated by two rapidly emerging aspects. First, more and more **web applications integrate machine learning (ML) components** at the code or at the service level to implement business functions or to protect web applications from abuse[1]. Unfortunately, existing techniques to detect vulnerabilities in web application lack the sophistication to interact with and interpret the behavior of ML components, thus impeding the analysis and testing of a larger and larger number of ML-based web applications. Second, web applications are facing more and more a social and political pressure to be designed, implemented, and deployed in a way that **preserves the users privacy**. This requirement is especially relevant in Europe after the adoption of the General Data Protection Regulation (GDPR) that aims, among other things, to establish a framework to guide organizations to manage citizens'—and by extension web applications users'—personal data. GDPR requires developers to extend existing risk-based and secure development methodologies with privacy-preserving solutions when designing and implementing the application software. It also introduces specific challenges to existing testing methodologies, as the privacy of an application depends on the composition of a large number of third-party services and libraries, which are often not under developers' control. Moreover, while it is reasonable to believe that developers have a clear incentive in fixing security vulnerabilities, it might be counterproductive for them to collect *less* user data. These two aspects combined result in the fact that privacy testing cannot be performed solely on the developers' side, but it also requires constant monitoring from the end-user perspective. To summarize, we can classify existing shortcomings around three main challenges:

(C1). Security testing of web-based applications is a task of ever growing complexity, which cannot sufficiently be addressed with the current set of technologies. The rapid inclusion of ML classifiers as part of (or as backend of) web applications further reduces the effectiveness of traditional white and black-box testing solutions.

[1] E.g., Google Cloud's Vision API https://cloud.google.com/vision or Microsoft Cognitive Services https://azure.microsoft.com/en-us/services/cognitive-services/

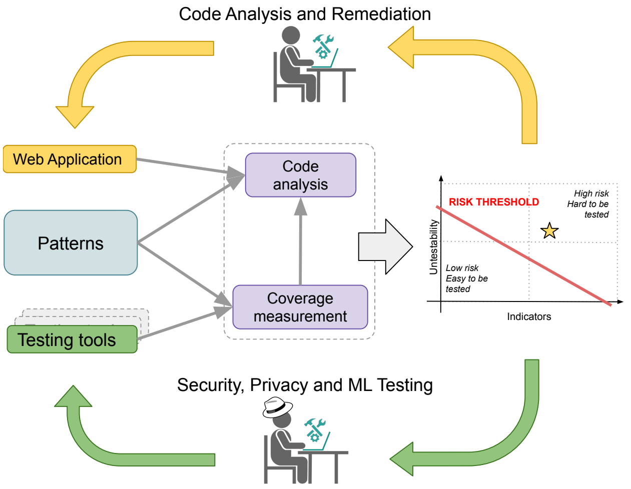


Figure 1: Our approach

(C2). Existing approaches do not provide a clear feedback to the developers on how to interpret the (lack of) results. In particular for areas of the application that are hard to test or even out of reach for the utilized testing tool, no information is given on the encountered problems or about potential remedies to provide more robust testing.

(C3). Existing static and dynamic frameworks to test web applications focus on the identification of well-known *security* vulnerabilities. *Privacy issues* are regularly overlooked and often left to a third party analyst to discover through tedious manual processes.

While there is a lot of research investigating over (C1), we believe that (C3) and (C2) tend to be overlooked and more research should focus on that. In our EU funded project TESTABLE [3] we are researching over all those challenges. Our idea is to re-design the way we test web applications around a new **testability metric**.

By applying this idea in the context of SAST we obtained very promising results [2]. In there, hundreds of tarpits—code instructions challenging for SAST tools—have been identified and captured in **testability patterns** that enabled us to measure SAST tools effectiveness, to make developers aware of the tarpits in their code (via automated discovery rules for the tarpits), and to improve the ”SAST testability” of open source web applications (via code refactoring routines removing the tarpits). Hundreds of new vulnerabilities have been uncovered after refactoring and confirmed by applications' owners.

An introduction to our approach, instantiated for the SAST domain, is discussed in Section 2. However, the approach is more general and future directions, shortly presented in Section 3, include its application to dynamic application security testing (DAST) and beyond security so to consider also the privacy and (adversarial) machine learning dimensions.

2. Approach and results for SAST

Fig. 1 depicts our approach. The core idea is to introduce the *(un-)testability dimension* in the risk score. Existing methodologies for measuring risk focus on the probability V that a certain web application would contain a vulnerability. These *mono-dimensional* methodologies are based on *vulnerability indicators* such as the size and complexity of the code, the type of information handled by the application, the number of commits over time, or the total number of developers—all elements that are difficult to modify in a running project. We propose instead a new orthogonal dimension measuring the **testability** of the application into the current notion of risk (cf. two-dimensional graph in the right-hand side of Fig. 1).

As a simple example, we can imagine two applications, A1 with a vulnerability indicator V=40% and A2 with V=63%. While lowering these values can improve the security and privacy of the applications, it can be very difficult to do that in practice. For instance, reducing code size or changing the number of developers are difficult aspects to act on. Moreover, it is possible that A1 (V=63%) avoids certain coding and practices (we refer to them as **testability patterns** in our work), thus making the application easier to analyze by testing tools. As a result, even if the likelihood of containing a vulnerability is higher for A1, the developers can expect that most of these problems will be detected during the testing phase—thus resulting in a final product that is more secure than the one corresponding to A2 (V=40%).

For space limitation, we illustrate in the next subsections the three main activities of our approach when applied to SAST, providing a summary of the results published in [2]. However the approach is more general and can be applied beyond SAST (see Section 3).

2.1. Testability patterns creation

Core to our research idea is the identification of the crucial testability patterns for specific testing approaches of web applications.

In the context of SAST, we created hundreds of testability patterns for both PHP and JavaScript (JS), two of the most popular web programming languages. A SAST testability pattern captures a few code instructions in language X that, when present, may impede the ability of state-of-the-art tools to analyze an application developed in X. Those few code instructions are referred to as the ”tarpit”. For instance, let us consider the following pattern instance for PHP:

```php
1 function F($var) {
2    return $var;
3 }
4 $a = $_GET["p1"]; // source
5 $b = call_user_func("F", $a); // tarpit
6 echo $b; // sink
```

First, note that all instances include, in purpose, a simple cross-site scripting (XSS) vulnerability to set an expected result when measuring SAST tools: in line 4 the attacker controls the input parameter `"p1"` (the source) that is printed, without any sanitization, into the HTML web page in line 6 (the sink). Second, in between this source-sink data-flow there is the *tarpit*. The tarpit is the code area that may confuse the SAST tools. The tarpit in line 5 is a form of *dynamic dispatching* (reflection) that allows the programmer to invoke a function specified by passing its name inside a string.

This specific pattern instance, that we will refer to as *simple dispatching*, is hardcoding a constant parameter `"F"`, thus making the target function resolvable from a static analysis perspective. Other similar pattern instances could be created. For instance, that constant could be first assigned to a variable and that variable could be then used as first parameter of `call_user_func`. Similarly, a concatenation instruction could be used as first parameter of the dynamic dispatching making the tarpit even more challenging and so on and so forth.

To be comprehensive, the internal specifications, and the APIs of both PHP and JS were reviewed and distilled into many SAST testability patterns, around several categories (e.g., object-oriented programming, security, etc): 120 instances were created for PHP and 150 for JS. We made all these patterns available to the entire web and SAST community [2].

2.2. Measurement and advancement of security, privacy and ML testing

Based on our patterns a comprehensive measurement of current testing approaches will be conducted to identify strenghts and weaknesses. Advanced techniques can be then designed and implemented to overcome some of these weaknesses (cf. lower green loop in Figure 1).

For SAST, we considered so far only the measurement part. An arsenal of 11 commercial and open-source SAST tools (6 for PHP and 5 for JS) were selected and measured against the created pattern instances. The best commercial tools were only able to handle 50% of the PHP and 60% of the JS tarpits, thus potentially leaving large parts of an application code unexplored. For instance, only 2 tools over 6 were able to detect the XSS in the simple dispatching pattern instance outlined above. The measurements, in [2] are detailed over the pattern categories showing that certain SAST tools may exceed in a category and fall short toward others. All these are precious information for SAST tools' owners and for the research community trying to advance SAST. You can know very precisely which pattern instances are blocking your tool and invest enginnering effort to support them in a future release.

2.3. Web Application Analysis and Remediation

For our risk measurements and iterative testing process, it is essential to reliably identify testability patterns in the source code of a web application under test. Thus, analysis techniques (mainly at the code level, but not only) will be designed to detect the testability patterns. Furthermore, following up on the results of our web application analysis, we will investigate remediation techniques such as code debloating and refactoring methods to remove problematic code patterns or replacing them with alternatives that are better handled by testing tools (cf. upper yellow loop in Figure 1).

In the context of SAST, to measure the impact on the unsupported tarpits, automated *discovery rules* were implemented and run to analyse more than 3000 open-source applications (the Testbed, in short). The experiments demonstrate that these tarpits are very common in the real world: the average project contains 21 different tarpits and even the best SAST tool cannot process more than 20 consecutive instructions without encountering a tarpit that prevents it from correctly analyzing the code. For instance, the dispatching pattern instance was discovered in more than 500 projects in the Testbed. The ability to automatically discover each tarpit brings many benefits. First, it can provide immediate and precise feedback to the developers about the tarpits in their code (e.g., by integrating the discovery rules into an IDE plugin). This information can then be used to make an informed decision about which combination of SAST tools are better suited to analyze the code, which parts of the application are *blind spots* for a static analyzer and thus may require a more extensive code review process, and which region of code could be *refactored* into *more testable* alternatives.

A few experiments were executed to evaluate the power of code refactoring as a mean to make an application more testable for SAST tools. By running SAST tools both before and after the transformations, a significant improvement in the overall testability of the application was observed. More than 400 new true positives emerged upon transformations and 188 have been already confirmed by the respective team, 55 of which affected very popular projects with more than 1000 stars in Github.

3. Conclusion and future directions

The outcomes from the previous section confirm the added-value of the overall approach of Figure 1, when applied to SAST, not only in measuring testing tools, but also and foremost in the impact of removing testability patterns' tarpits as a means to increase the testability of web applications.

Besides maturing the work for SAST (e.g., by computing also vulnerability indicators and combining that with testability to derive an overall risk score), we aim to apply our approach to DAST, privacy and machine learning. The core idea is to create testability patterns for all these areas so to measure the *available* state-of-the-art testing tools, advance/develop the techniques underlying these tools, and mitigate these patterns whenever possible to increase the testability of the application. For instance, we expect many testability patterns to emerge as a means to probe DAST crawlers. A failure in the crawling phase may leave a large portion of the web application uncovered. On the other hand, aware that privacy testing, when compared to security testing, is still in its infancy, our research in that area will focus on defining the scope of privacy testing and implement privacy testing techniques to address relevant and well-known web related privacy issues. Last, but not least, we want to create testability patterns which are hindering security and privacy testing of ML-based components so to measure these patterns against the many emerging tools to detect adversarial machine learning attacks.

References

[1] Edgescan. 2020 vulnerability statistics report. 2020.

[2] Feras Al Kassar, Giulia Clerici, Luca Compagna, Fabian Yamaguchi, and Davide Balzarotti. Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications. In *Network and Distributed System Security (NDSS) Symposium*, NDSS 22, April 2022.

[3] TESTABLE consortium. TESTABLE: TestabiliTy Pattern-driven Web Application Security and Privacy Testing. https://testable.eu/, Accessed April 27, 2022.

TESTABLE

Testability-driven Security and Privacy Testing for Web Applications

testable.eu

VISION

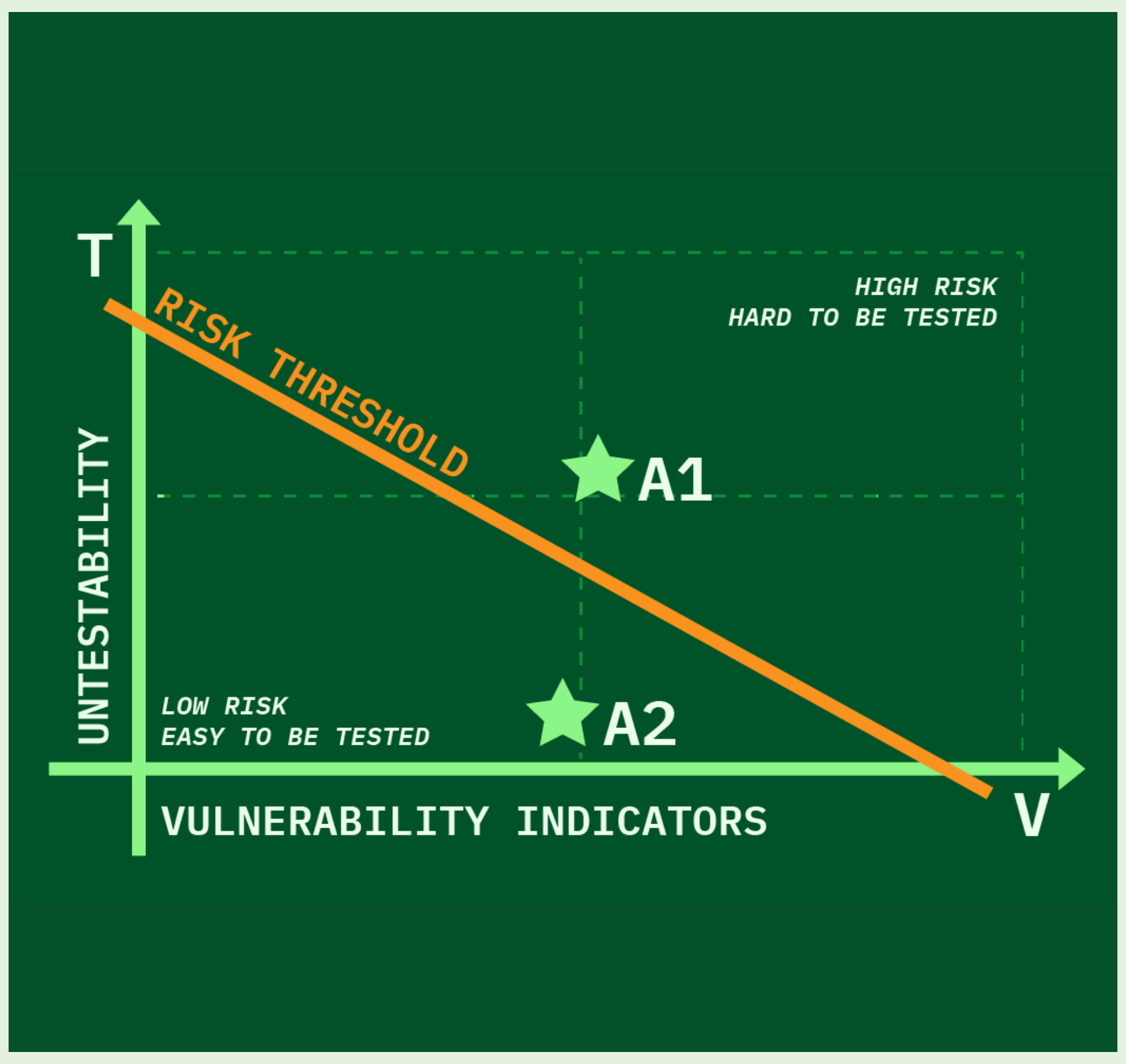
A New Testability Metric

- Precise risk score (probability that a web app contains issues)
- New two-dimensional space that not only includes existing indicators (e.g., LoC, presence of security-sensitive function calls, …)
- Novel dimension of testability of the application with respect to a certain class of testing techniques (e.g., SAST, DAST, …)
- Compute testability via measurable, discoverable and (possibly) transformable testability patterns capturing challenges for a class of testing techniques

A New Decision and Action Space

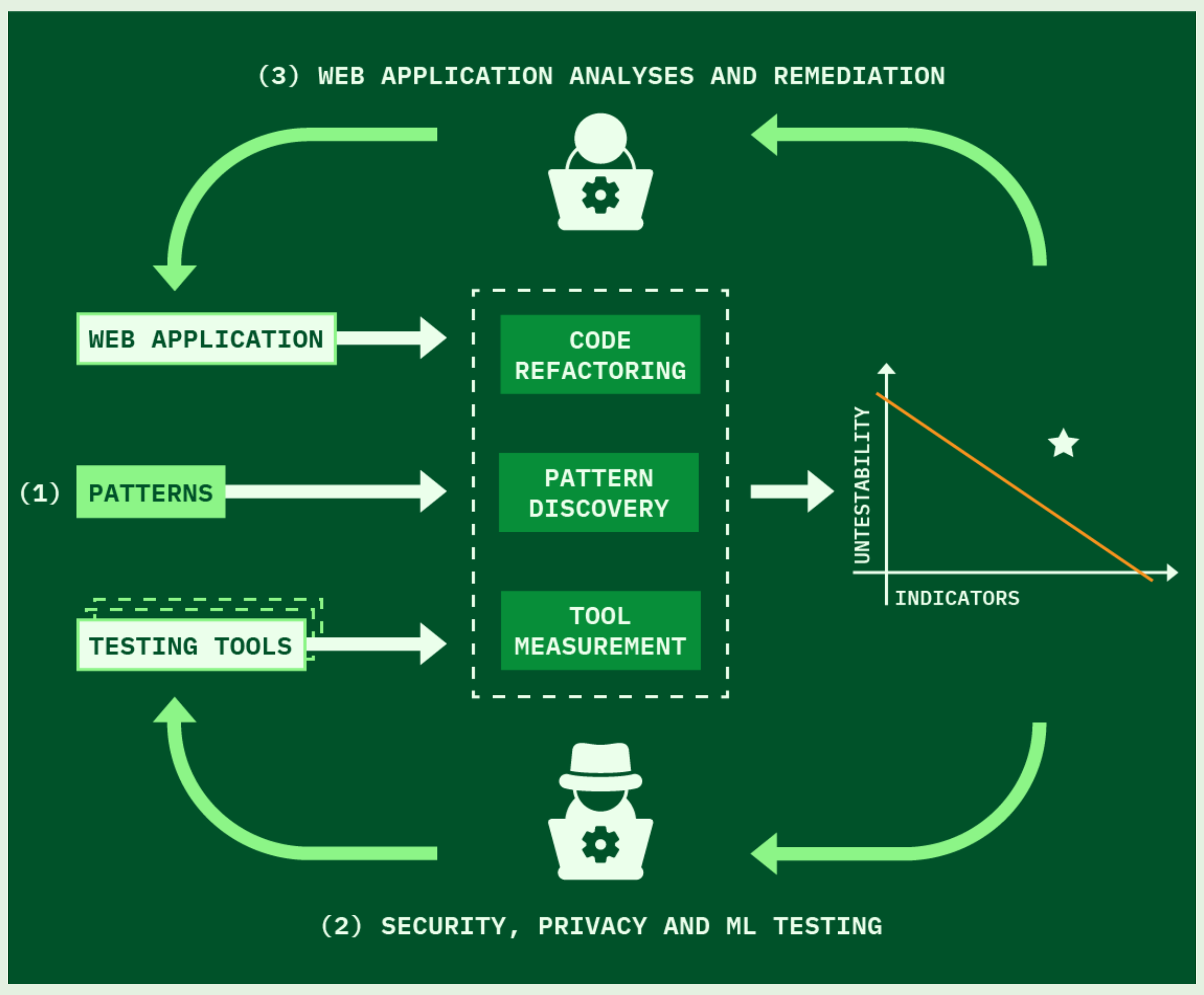
Testability metric provides a natural way to improve the security and privacy of web applications over time via three actions

- Optimize testing strategies by focusing on the problematic components, by reconfiguring existing tools, or procuring new ones
- Review design and code to increase application testability, e.g., by refactoring the program to remove testability patterns
- Deployment or development of additional defense in-depth layers when no other actions are viable



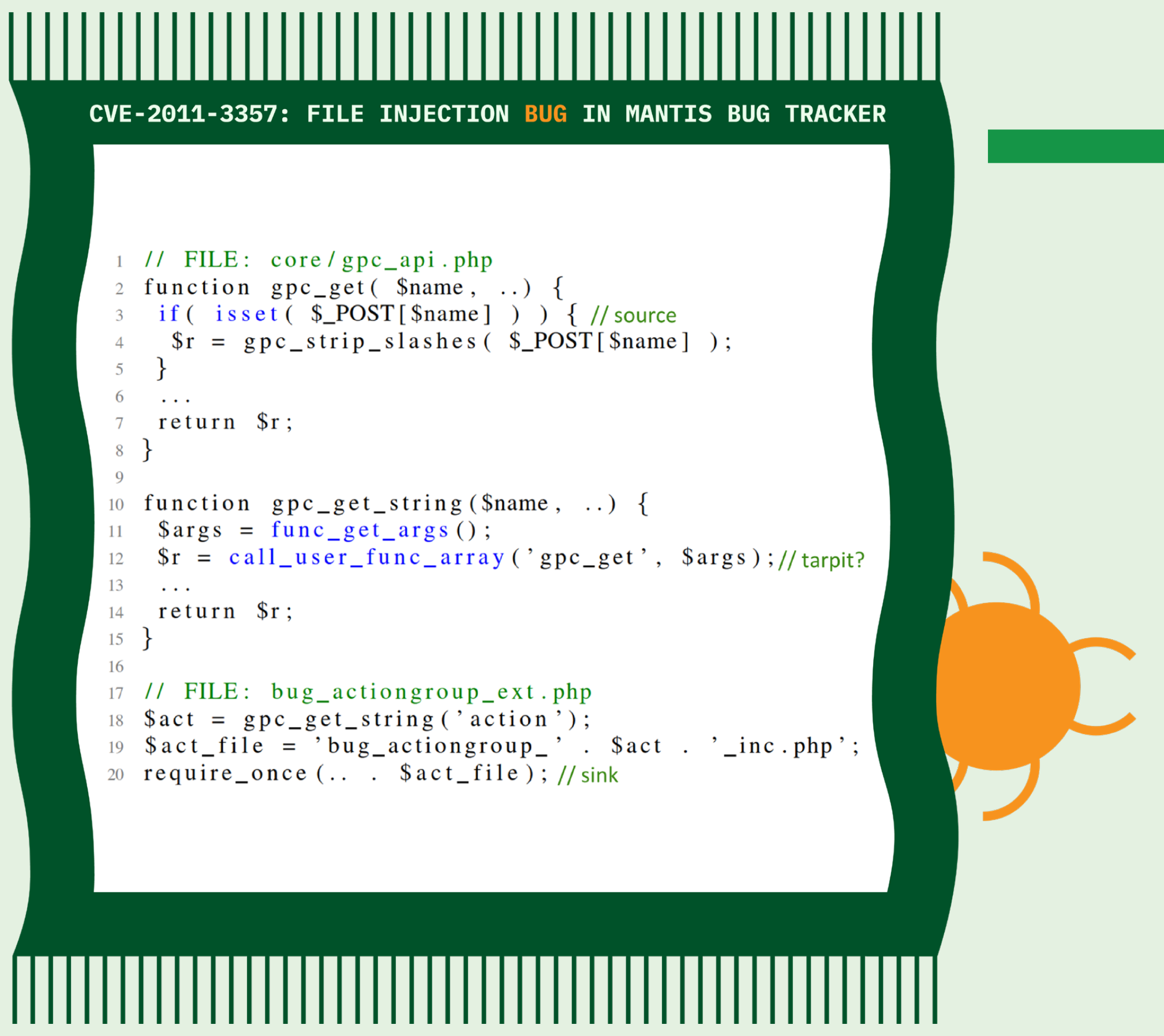
In-scope Domain Areas

- Security: improve security testing techniques and strategies (SAST, DAST, and HAST)
- Privacy: design novel static and dynamic techniques to test for privacy-related problems
- ML: develop new techniques to enable security and privacy testing of AI/ML modules within web apps

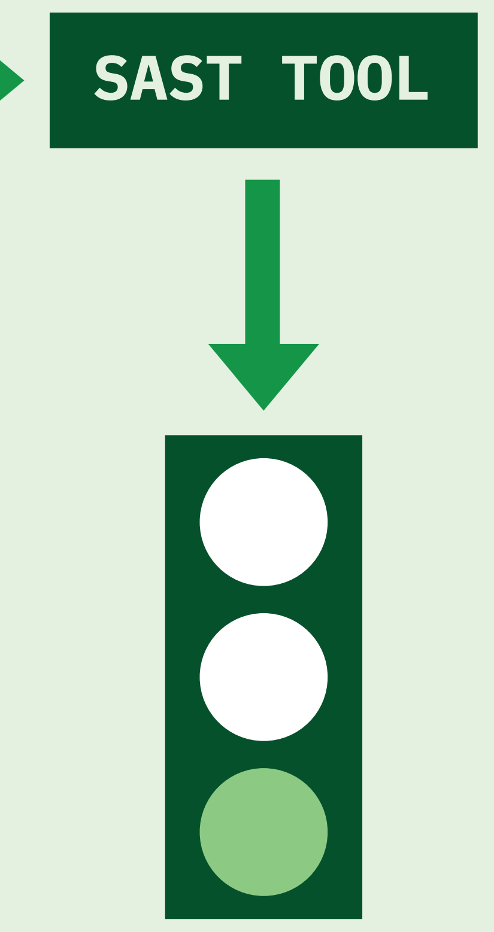


TESTABLE OVER SAST: A FIRST STORY

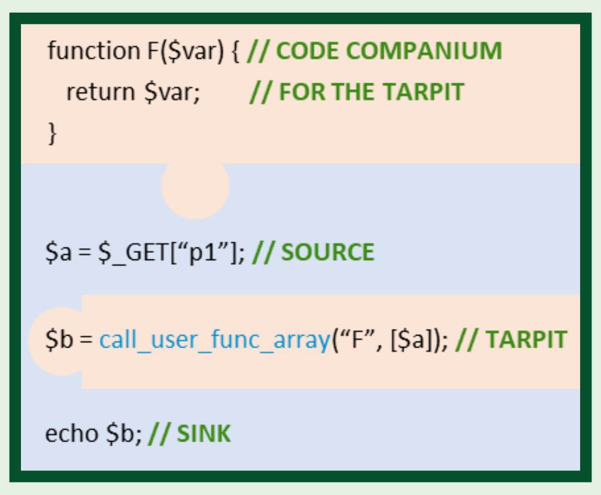
No bugs under the carpet? Testability for SAST: good?


CVE-2011-3357: FILE INJECTION BUG IN MANTIS BUG TRACKER

```
1  // FILE: core/gpc_api.php
2  function gpc_get( $name, ..) {
3   if( isset( $_POST[$name] ) ) { // source
4    $r = gpc_strip_slashes( $_POST[$name] );
5   }
6   ...
7   return $r;
8  }
9
10 function gpc_get_string($name, ..) {
11  $args = func_get_args();
12  $r = call_user_func_array('gpc_get', $args);// tarpit?
13  ...
14  return $r;
15 }
16
17 // FILE: bug_actiongroup_ext.php
18 $act = gpc_get_string('action');
19 $act_file = 'bug_actiongroup_' . $act . '_inc.php';
20 require_once(.. . $act_file); // sink
```



Pattern creation (1)



```
function F($var) { // CODE COMPANIUM
  return $var;      // FOR THE TARPIT
}

$a = $_GET["p1"]; // SOURCE

$b = call_user_func_array("F", [$a]); // TARPIT

echo $b; // SINK
```

SAST measurement (2)

SAST	*CORRECT*
RIPS	NO
phpSAFE	NO
WAP	NO
Progpilot	YES
Comm1	NO
Comm2	YES

Created Many Patterns/Tarpits

- PHP: ~120 pattern instances
- JS: ~150 pattern instances

SAST Tools Measurement: A lot to Improve

- PHP: <50% support rate
- JS: ~60% support rate

Our Tarpits are Highly Prevalent

- created tarpit discovery rules
- analyzed >3000 PHP apps
- in AVG: 21 tarpit per app, one tarpit every 20 LoC

Refactoring Tarpits Increase Application Testability

- two exps: manual and automated refactoring
- >400 new vulnerabilities emerged upon refactoring (~200 already confirmed)

Pattern discovery and transformation (3)

```
// SAST tools detect File
// injection, by replacing
// the tarpit discovered
// in line 12 with:
12    $r = gpc_get($args);
```

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY
EURECOM Sophia Antipolis
Technische Universität Braunschweig
uc3m
SAP
ShiftLeft
IMQ MINDED SECURITY
NortonLifeLock
Pluribus One seeing one in many