MCP: A Security Testing Tool Driven by Requirements

Phu X. Mai, Fabrizio Pastore, Arda Goknil, Lionel C. Briand
SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
{xuanphu.mai, fabrizio.pastore, arda.goknil, lionel.briand}@uni.lu

***Abstract*—We present MCP, a tool for automatically generating executable security test cases from misuse case specifications in natural language (i.e., use case specifications capturing the behavior of malicious users). MCP relies on Natural Language Processing (NLP), a restricted form of misuse case specifications, and a test driver API implementing basic utility functions for security testing. NLP is used to identify the activities performed by the malicious user and the control flow of misuse case specifications. MCP matches the malicious user's activities to the methods of the provided test driver API in order to generate executable security test cases that perform the activities described in the misuse case specifications. MCP has been successfully evaluated on an industrial case study.**

***Index Terms*—System Security Testing, Natural Language Requirements, Natural Language Processing (NLP).**

I. Introduction

Software security has become a major concern in software development, starting from requirements analysis to implementation and testing. Security requirements focus on both security properties of the system and potential security threats [1]. For instance, in use case-driven methodologies [2], [3], security use cases describe security properties of the system (e.g., user authentication) while misuse cases describe malicious activities (e.g., bypassing the authorization schema). Security testing is driven by requirements [4] and, consequently, can be divided into two categories [5]: (1) security functional testing validating the specified security properties, and (2) security vulnerability testing addressing the identification of system vulnerabilities. Though several security testing approaches exist [5], the automated generation of security test cases from Natural Language (NL) security requirements remains limited in industrial settings.

We present a tool, MCP, which generates security vulnerability test cases from misuse case specifications in NL. MCP borrows some concepts from *natural language programming*, a term which refers to approaches automatically generating software programs (e.g., executable test cases) from NL specifications [6], [7]. MCP assumes that security requirements are elicited according to a misuse case template including keywords to support the extraction of control flow information. It requires a test driver API for basic security testing activities (e.g., requesting a URL).

Using Natural Language Processing (NLP) techniques, MCP extracts the control flow of malicious activities described in misuse cases. It translates the extracted control flow into sequences of executable instructions (e.g., invocations of the test driver API's functions) that implement the malicious activities. To this end, we adapt the idea, followed by other works [8], [9], [10], [11], of combining string similarity and ontologies [12]. Similarly, MCP builds an ontology that captures the structure of the given test driver API. It generates executable instructions by looking for nodes in the ontology that are similar to phrases in NL requirements. The innovative idea behind MCP is that it integrates additional analyses required to enable automated testing, i.e., the identification of test inputs and the generation of input values and test oracles.

In the remaining sections, we outline MCP's features and components. We highlight the findings from our evaluation of MCP over an industrial case study.

II. Related Work

Most security testing approaches and tools focus on a particular vulnerability (e.g., buffer overflows [13] and code injection vulnerabilities [14]). They deal with the generation of simple inputs (e.g., strings, files), and cannot be adopted for complex attack scenarios involving several interactions among parties. Model-based tools are capable of generating test cases for such complex attack scenarios [15] but require formal models, which limit their adoption in industry where security requirements are mostly in NL.

There are approaches generating functional system test cases from NL requirements [16], [17], [18], [19]. However, these approaches can be adopted in the context of security functional testing, but not security vulnerability testing since they generate test cases only for the intended behavior of the system. In contrast, security vulnerability testing deals with the problem of simulating the behavior of a malicious user.

MCP can generate security vulnerability test cases for complex attack scenarios without any formal security model. It employs natural language programming and NLP techniques to automatically generate those test cases from misuse cases.

III. Tool Overview

MCP is the tool supporting our approach for automatically generating security vulnerability test cases from misuse case specifications, described in a recent conference paper [20]. Fig. 1 presents an overview of our tool. In Step 1, the user manually elicits misuse cases according to the Restricted Misuse Case Modeling (RMCM) template [2] that includes keywords to support the extraction of control flow information.

This paper appears in the Companion proceedings of the 41st International Conference on Software Engineering (ICSE2019). IEEE.

Once security requirements are captured in the form of misuse cases, MCP automatically checks whether the misuse case specifications conform to the RMCM template. If any inconsistency is detected, the tool reports these inconsistencies.

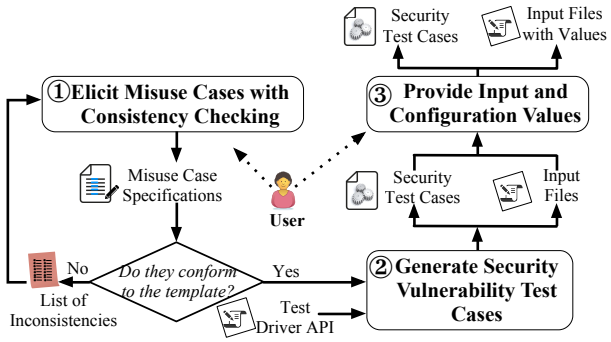

Fig. 1. Tool Overview

In Step 2, MCP processes the misuse case specifications once they are deemed consistent. It automatically generates executable test cases and test input files from the misuse case specifications and a test driver API that implements basic security testing activities (e.g., requesting a URL or starting a network sniffing tool). In Step 3, the user provides input values for the generated test input files to be used during testing. In the rest of this section, we elaborate on each step in Fig. 1.

A. Elicitation of Misuse Cases

The user elicits misuse cases based on the RMCM method. RMCM extends the Restricted Use Case Modeling (RUCM) method [21], [17], [22], [23], [24] to support the specification of security requirements in an analyzable form [2].

RMCM provides a template that characterizes basic and alternative flows in misuse cases. A basic threat flow describes a nominal scenario for a malicious actor to harm the system (Lines 4 - 13 in Fig. 2). It contains misuse case steps and a postcondition. Alternative flows capture failed attacks (e.g., Lines 21 - 26), while alternative *threat* flows describe alternative attack scenarios. For instance, in Lines 14 - 20, the specific alternative threat flow *SATF1* describes another successful attack scenario where the resource contains a role parameter. A specific alternative flow always depends on a specific step of the reference flow, specified by the `RFS` keyword. The `IF .. THEN` keyword describes the conditions under which alternative flows are taken (e.g., Line 16). The RMCM keywords are used to specify actors (e.g., `MALICIOUS user`), successful attacks (e.g., `EXPLOIT`), attack patterns (e.g., `PROVIDE SQLI VALUES`) and control flow (e.g., `FOREACH`). All other terms in a specification are freely selected by the user.

MCP is provided with a set of predefined misuse case specifications derived from the OWASP testing guidelines [4]. They can be reused (or adapted) across projects; in addition, the user can write new, system specific misuse cases.

B. Security Vulnerability Test Case Generation

MCP takes as input a test driver API and misuse case specifications to automatically generate security vulnerability test cases. The test driver API is used by the generated test cases to execute the functions of the system under test. MCP is released with a general test driver API in Python that can be used to test any Web system; if needed, the user can extend the test driver API or use another one (e.g., one for functional testing). Misuse cases drive the generation of the test code.

1	**MISUSE CASE** Bypass Authorization Schema
2	**Description** The MALICIOUS user accesses resources that are dedicated to a user with a different role.
3	**Precondition** For each role available on the system, the MALICIOUS user has a list of credential of users with that role, plus a list functions/resources that cannot be accessed with that role.
4	**Basic Threat Flow**
5	1. FOREACH role
6	2. The MALICIOUS user sends username and password to the system through the login page
7	3. FOREACH resource
8	4. The MALICIOUS user requests the resource from the system.
9	5. The system sends a response page to the MALICIOUS user.
10	6. The MALICIOUS user EXPLOITS the system using the response page and the role.
11	7. ENDFOR
12	8. ENDFOR
13	**Postcondition:** The MALICIOUS user has executed a function dedicated to a user with different role.
14	**Specific Alternative Threat Flow (SATF1)**
15	RFS 4.
16	1. IF the resource contains a role parameter in the URL THEN
17	2. The MALICIOUS user modifies the role values in the URL.
18	3. RESUME STEP 4.
19	4. ENDIF.
20	**Postcondition:** The MALICIOUS user has modified the URL.
21	**Specific Alternative Flow (SAF1)**
22	RFS 6
23	1. IF the response page contains an error message THEN
24	2. RESUME STEP 7.
25	3. ENDIF.
26	**Postcondition** The malicious user cannot access the resource dedicated to users with a different role.

Fig. 2. 'Bypass Authorization Schema' misuse case specification.

MCP generates, for each misuse case, an executable test case, a JSON test input file without input values (see Fig. 3), and some configuration files for the test driver API (see Fig. 4).

Configuration files are used to configure general purpose test API methods for a specific system. For Web systems, these methods send inputs to the system through a Web page. MCP automatically generates configuration files for these methods.

Each generated test case is a Python class implementing a `run` method executing the malicious activities described in a given misuse case. We chose Python since it is a popular language for Web systems. Fig. 5 shows part of the test case generated from the misuse case in Fig. 2. MCP declares and initializes the variables `system`, `maliciousUser` and `inputs` (Lines 3 - 5). The variable `system` refers to an instance of the class `System` whose methods trigger the functions of the system under test (e.g., `request`). The variable `maliciousUser` refers to the test class simulating the malicious user behavior. The variable `inputs` refers to

```
{"role": [
  {
    "password": "REPLACE-THIS-STRING",
    "role": "REPLACE-THIS-STRING",
    "username": "REPLACE-THIS-STRING"
    "resource": [
    {
      "resource": "REPLACE-THIS-STRING",
      "error_message": "REPLACE-THIS-STRING",
      "role_values": "REPLACE-THIS-STRING",
      "the_resource_contains_the_role_parameter_in_the_URL": "PUT-EXECUTABLE-CODE",
      "the_resource_contains_the_role_parameter_in_the_HTTP_post_data": "PUT...
    },
     ADD-MORE-ENTRIES
    ],
  },
  ADD-MORE-ENTRIES
  ]
}
```

Fig. 3. Input file generated by MCP.

```
"URL" : "http://link_to_the_page",
"Position" : "Choose POST-data, URL or BOTH",
"username" : "USERNAME_id_in_the_page",
"password" : "PASSWORD_id_in_the_page",
"Others" : ["list_of_other_parameters"]
```

Fig. 4. Configuration file generated to send inputs through to the login page.

```
1  class bypassAuthorizationSchema(HTTPTester):
2    def run(self):
3      system = System(path=self.rootPath)
4      maliciousUser = self
5      inputs = self.loadInput("inputs.json")
6      roleIter = inputs["role"].__iter__()
7      while True:
8        try:
9          role = roleIter.__next__()
10         parameters = dict()
11         parameters["password"] = role["password"]
12         parameters["username"] = role["username"]
13         system.send("login page",parameters)
14         resourceIter = role["resource"].__iter__()
15         while True:
16           try:
17             resource = resourceIter.__next__()
18             if not eval(resource["the_resource_contains_a_role_
19                 parameter_in_the_URL"]):
20               if not eval(resource["the_resource_contains_a_role_parameter..
21                 system.request(resource)
22                 maliciousUser.responsePage = system.responsePage
23                 if not responsePage.contains( resource["error message"] )
24                   parameters = dict()
25                   parameters["resource"] = resource["resource"]
26                   parameters["role"] = role["role"]
27                   system.exploit(parameters)
28                 else:
29                   maliciousUser.abort("The MALICIOUS user CANNOT ex...")
```

Fig. 5. Part of the test case generated from the misuse case in Fig. 2.

a dictionary populated with input values in the JSON input file. The instructions in the test case (e.g., a call to an API method) are selected based on string similarity between the phrases in the misuse case steps and the variables, methods and parameters of the test driver API.

C. Providing Input and Configuration Values

The user provides input values for the generated JSON input file. The generated input file reflects the relations between input entities in the misuse case (see Fig. 3). Fig. 6 shows example input values for the JSON input file in Fig. 3.

Simple inputs are represented as key-value pairs; complex inputs are given as a list of dictionaries (e.g., `role` in Fig. 3). A complex input is generated every time an input entity (e.g., `role`) is referred to in an iteration (i.e., a step containing the keyword `FOREACH`) because the body of iterations typically contains activities that provide a set of related inputs to the system (e.g., `username` and `password`). Every input entity referred to in the body of the iteration is captured by a key-value pair (e.g., the case for the key `username`). Nested iterations are captured by nested lists of dictionaries (e.g., the case of `resources`), since they usually describe groups of activities working on additional sets of related inputs. A special type of input are the values evaluated in conditional statements in specifications (e.g., the case of `the_resource_contains...`) as, for these values, the user can provide either a boolean value (e.g., `TRUE`) or a Python expression to be evaluated at runtime.

```
{"role": [
    {
      "role": "Doctor",
      "username": "phu@mymail.lu"
      "password": "testPassword1",
      "resource": [
       {
         "resource": "http://www.icare247.eu/?q=micare_invite&accountID=11"
         "error_message": "error",
           "the_resource_contains_the_role_parameter_in_the_URL": False,
       },
      {
        "resource": "http://www.icare247.eu/?q=micare_skype/config&clientID=36"
...
      }, ], }
    {
      "role": "Patient",
    . . .
```

Fig. 6. Part of the JSON file in Fig. 3 with input values.

```
"URL" : "http://www.icare247.eu/?q=micare_user_login",
"Position" : "POST-data",
"password" : "passwd",
"username" : "email",
"Others" : ["op", "form_build_id", "form_id"]
```

Fig. 7. Values provided to the configuration file in Fig.4

Configuration files are generated to match input entities in misuse cases and input parameters of Web pages of the system (see Fig. 4). Fig.7 shows the values provided to the configuration file in Fig. 4.

The test case is executed using the Python interpreter. During execution, the test case loads the inputs and configuration values from the JSON files and invokes the test driver API functions when needed. Vulnerabilities are reported by the test driver API method `exploit` (Line 27 in Fig. 5).

IV. Implementation & Availability

MCP has been implemented as a Java application. Fig. 8 shows the architecture of MCP.

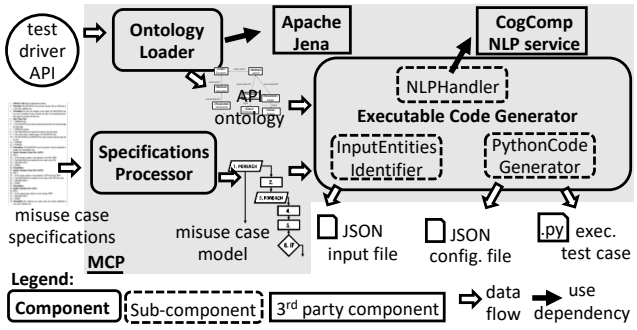

Fig. 8. MCP Architecture

SpecificationProcessor parses misuse case specifications to extract a model (misuse case model) that captures the control flow in the specifications. *OntologyLoader* maps the test driver API into an OWL ontology to have a structured representation of the API elements (i.e., classes, methods, and parameters).

ExecutableCodeGenerator identifies input entities and generates executable code. *NLPHandler* executes NLP on each misuse case step. More specifically, *NLPHandler* executes the *CogComp NLP pipeline* [25] to perform Semantic Role Labelling (SRL), an advanced NLP procedure that identifies roles of phrases in sentences (e.g., an actor performing an activity). *InputEntitiesIdentifier* uses SRL results to identify input entities. To speed up NLP, *NLPHandler* relies on the CogComp NLP pipeline running as a Web service[1]. If the service is not reachable, the analysis is executed locally.

PythonCodeGenerator generates Python code together with configuration files. It processes the misuse case model and generates a method call or an assignment for each use case step, except condition and iteration steps which are translated into Python instructions. In general, thanks to SRL outputs, a method call is identified by selecting a method (1) that belongs to a class instance with a name similar to either the actor performing the activity or the destination in the sentence, (2)

[1]MCP can rely both on the installation of the University of Pennsylvania, http://macniece.seas.upenn.edu:4001/annotate, or services running in-house.

that has a name textually similar to the verb in the sentence, and (3) that has parameters matching the remaining semantic roles in the sentence. An assignment instruction is generated when some data is exchanged between an actor and the system. It is generated by looking for two variables textually similar to the source and destination of the data exchange.

Additional details about MCP, including executable files and a screencast, are available on the tool's website at:

https://sntsvv.github.io/MCP/

V. Empirical Evaluation

We evaluated the feasibility of MCP with an industrial case study, which is a healthcare software system developed in the context of the EU project EDLAH2 [26]. The EDLAH2 system is a representative example of a modern, user-oriented system including Web, mobile and wearable user interfaces.

The EDLAH2 engineers followed the RMCM methodology to capture security requirements. The EDLAH2 misuse case specifications include a total of 68 misuse cases which describe both general attack patterns derived from the OWASP guidelines [4], [27] and system specific attacks that leverage some characteristics of the EDLAH2 system.

We used MCP to generate executable test cases from 12 misuse case specifications. We selected 12 misuse cases targeting the Web interface and with the highest risk according to the OWASP risk rating methodology [4].

MCP successfully generated one executable test case for each specification. Nine of the generated test cases identified real vulnerabilities affecting the system. The generated test cases do not contain any programming errors despite the generated code being not trivial (791 lines of code in total, 172 method calls, 44 assignments, and 260 method arguments). Also, MCP successfully identified the input entities required to execute the test cases, with a high precision and recall of 0.97 and 0.91, respectively.

VI. Conclusion

We presented a tool, MCP, that automatically generates executable security vulnerability test cases from misuse case specifications. The key characteristics of our tool are (1) the use of security requirements in NL, only involving a few keywords, (2) the automated translation of these requirements into executable test cases targeting vulnerabilities, and (3) the automated identification of test inputs. We evaluated MCP over an industrial Web application. The evaluation shows that our tool is able to generate test cases that are effective at identifying vulnerabilities. MCP also reduces the effort required to perform security vulnerability testing. Ongoing work regards the adoption of MCP to test Android systems and the automated generation of the inputs required by MCP.

Acknowledgment

This work is supported by the National Research Fund, Luxembourg INTER/AAL/15/11213850 and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277).

References

[1] B. Fabian, S. Gurses, M. Heisel, T. Santen, and H. Schmidt, "A comparison of security requirements engineering methods," *RE'10*, vol. 15, pp. 7–40.

[2] P. X. Mai, A. Goknil, L. K. Shar, F. Pastore, L. C. Briand, and S. Shaame, "Modeling security and privacy requirements: a use case-driven approach," *Information and Software Technology*, vol. 100, pp. 165–182, 2018.

[3] A. L. Opdahl and G. Sindre, "Experimental comparison of attack trees and misuse cases for security threat identification," *Information and Software Technology*, vol. 51, pp. 916–932, 2009.

[4] M. Meucci and A. Muller, "OWASP Testing Guide v4," https://www.owasp.org/images/1/19/OTGv4.pdf.

[5] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Chapter one - security testing: A survey," ser. Advances in Computers. Elsevier, 2016, vol. 101, pp. 1–51.

[6] B. W. Ballard and A. W. Biermann, "Programming in natural language: "nlc" as a prototype," in *ACM'79*, 1979, pp. 228–237.

[7] O. Pulido-Prieto and U. Juárez-Martínez, "A survey of naturalistic programming technologies," *ACM Computing Surveys*, vol. 50, no. 5, pp. 70:1–70:35, 2017.

[8] C. Manning, M. Surdeanu, J. Bauer, J. abd Finkel, S. Bethard, and D. McClosky, "The stanford CoreNLP natural language processing toolkit," in *ACL'14*, 2014, pp. 55–60.

[9] V. Le, S. Gulwani, and Z. Su, "SmartSynth: Synthesizing smartphone automation scripts from natural language," in *MobiSys'13*, 2013.

[10] D. Guzzoni, C. Baur, and A. Cheyer, "Modeling human-agent interaction with active ontologies," in *AAAI Spring Symposium'07*, 2007, pp. 52–59.

[11] M. Landhausser, S. Weigelt, and W. F. Tichy, "NLCI: a natural language command interpreter," *Automated Software Engineering*, vol. 24, no. 4, pp. 839–861, 2017.

[12] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199 – 220, 1993.

[13] S. Ognawala, M. Ochoa, A. Pretschner, and T. Limmer, "MACKE: Compositional analysis of low-level vulnerabilities with symbolic execution," in *ASE'16*, 2016, pp. 780–785.

[14] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: An input mutation approach," in *ISSTA'14*, 2014, pp. 259–269.

[15] F. Lebeau, B. Legeard, F. Peureux, and A. Vernotte, "Model-based vulnerability testing for web applications," in *ICSTW'13*, 2013.

[16] G. Carvalho, D. Falcão, F. Barros, A. Sampaio, A. Mota, L. Motta, and M. Blackburn, "NAT2TESTSCR: Test case generation from natural language requirements based on SCR specifications," *Science of Computer Programming*, vol. 95, pp. 275–297, 2014.

[17] C. Wang, F. Pastore, A. Goknil, L. C. Briand, and M. Z. Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *ISSTA'15*, 2015, pp. 385–396.

[18] ——, "UMTG: a toolset to automatically generate system test cases from use case specifications," in *ESEC/FSE'15*, 2015, pp. 942–945.

[19] M. Kaplan, T. Klinger, A. M. Paradkar, A. Sinha, C. Williams, and C. Yilmaz, "Less is more: A minimalistic approach to UML model-based conformance test generation," in *ICST'08*, 2008, pp. 82–91.

[20] P. X. Mai, F. Pastore, A. Goknil, and L. C. Briand, "A natural language programming approach for requirements-based security testing," in *ISSRE'18*, 2018, pp. 58–69.

[21] T. Yue, L. C. Briand, and Y. Labiche, "Facilitating the transition from use case models to analysis models: Approach and experiments," *ACM TOSEM*, vol. 22, no. 1, pp. 1–38, 2013.

[22] I. Hajri, A. Goknil, L. C. Briand, and T. Stephany, "Applying product line use case modeling in an industrial automotive embedded system: Lessons learned and a refined approach," in *MODELS'15*, 2015, pp. 338–347.

[23] ——, "Configuring use case models in product families," *Software and System Modeling*, vol. 17, no. 3, pp. 939–971, 2018.

[24] ——, "PUMConf: a tool to configure product specific use case and domain models in a product line," in *FSE'16*, 2016, pp. 1008–1012.

[25] University of Pennsylvania, "CogComp NLP," http://cogcomp.org, 2018.

[26] "EDLAH2: Active and Assisted Living Programme," http://www.aal-europe.eu/projects/edlah2/.

[27] "OWASP, Android Testing Guidelines." https://www.owasp.org/index.php/Android_Testing_Cheat_Sheet.