A Meta-model for Software Protections and Reverse Engineering Attacks : an instance of the meta-model

C. Basile[a], D. Canavese[a], L. Regano[a], P. Falcarin[b,*], B. De Sutter[c]

[a] *Dipartimento di Automatica e Informatica, Politecnico di Torino, Italy*
[b] *Department of Computing and Engineering, University of East London, United Kingdom*
[c] *Computer Systems Lab, Department of Electronics and Information Systems, Ghent University, Belgium*

Abstract

This supplementary document presents an additional use case analysis on top of the paper titled "A Meta-model for Software Protections and Reverse Engineering Attacks" by the same authors, in the form of an instance of the meta-model filled in with the data from the risk analysis and mitigation of an open source software application.

1. Introduction

Identifying the threats against a software application is an indispensable task to mitigate the risk that attackers are able to compromise the assets therein. Automating this task would have an tremendous impact in helping software developers, who usually have limited or no competence in software protection, to achieve a reasonable level of protection of their assets. However, such an automation is impossible without a proper formalization of the knowledge needed to perform this activity.

We have proposed a meta-model able to describe all the entities involved in the risk analysis and mitigation. The meta-model allows describing all the relevant parts of an application (data, functions, pieces of code), the assets, the attacks against each application part and the protections that mitigate these attacks [1]. The meta-model also describes solutions, which describe how a combination of protections can be applied on each application part that deserves to be protected. This meta-model has been designed to be the foundation of a knowledge base that allows reasoning about the risk analysis and mitigation process.

The following sections present an instance of the meta-model built on data from the risk analysis and mitigation of Sumatra[1]. This document presents the data that can be

*Corresponding author: Paolo Falcarin
Email addresses: cataldo.basile@polito.it (C. Basile), daniele.canavese@polito.it (D. Canavese), leonardo.regano@polito.it (L. Regano), falcarin@uel.ac.uk (P. Falcarin), bjorn.desutter@ugent.be (B. De Sutter)

[1] This example has been based Sumatra version 1.0.31, available here https://git.metabarcoding.org/obitools/sumatra/wikis/home

represented by the meta-model and serves as validation of the meta-model effectiveness in representing the data needed for risk analysis and mitigation.

This document is structured as follows. Section 2 presents more information about the application we have protected and how we have selected it. Section 2 presents the risk analysis and mitigation work flow we have executed to instantiate the meta-model. Section 4 describes the application parts in the Sumatra application and the assets we have selected. Section 5 presents the attacks that threaten the selected assets. Section 6 illustrates the protections that can be used to mitigate the risks against the assets and block/delay the identified attacks. Finally, Section 7 draws conclusions.

2. The use case

In the following sections we are going to present an instance of the meta-model for the protection of the Sumatra application, a C console application used to compare DNA sequences. Sumatra can be used either to compare all the sequences contained in a single dataset, or to perform a pairwise comparison of sequences contained in two datasets. While Sumatra is an open-source application, we have simulated its protection like it was a commercial software, whose comparison algorithm must be safeguarded against reverse engineering.

The DNA comparison is performed in four consecutive phases. First, the application parses the command line arguments, and depending on them calls the appropriate functions to compare the given DNA sequences. Then, the latter are parsed and stored in various data structures. These data structures are then used by the core algorithms to perform the actual comparison of the sequences. Finally, the results of the comparison are returned to the user.

We have identified in total 25 functions as assets. Table 1 lists them per phase. To show how the meta-model can be useful in protecting the application, we aim to safeguard all the assets from attacks against their confidentiality and integrity.

Full details about the meta-model instantiation we are going to present in the following sections is available as an ontology file[2], written in the Web Ontology Language 2 (OWL2).

[2] `https://github.com/SPDSS/Software-Protection-Meta-Model/blob/master/sumatra-model.owl`

Sumatra phase	Asset names	# assets
1	main	1
2	seq_readAllSeq2, cleanDB, addCounts, seq_fillHeader, seq_fillSeqOnlyATGC, seq_fillSeq, seq_fillDigitSeq, seq_readNextFromFilebyLine, uniqSeqsVector	9
3	compare1, calculateMaxAndMinLen, compare2, calculateMaxAndMinLenDB	4
4	seq_findSeqByAccId, seq_printSeqs printOnlySeqFromFastaSeqPtr, sortSeqsWithCounts, seq_getNext, reverseSortSeqsWithCounts, printHeaderAndSeqFromFastaSeqPtr, printOnlySeqFromChar, printResults, printOnlyHeaderFromFastaSeqPtr, printOnlyHeaderFromTable	11

Table 1: Functions marked as assets, grouped by Sumatra phases.

3. Work flow for risk analysis and mitigation of software application

The meta-model instance we will analyse in the following sections has been populated in a semi-automated way, by using a tool chain and corresponding work flow developed to assist a software developer in protecting their applications. In particular, the tools used include:

1. a parser of the application source code;
2. an automatic attack discovery tool;
3. an automatic protection discovery tool;
4. an tool that estimates of software metrics after the application of protections to the source code;
5. a tool to hide the protected assets.

First, the application source code is parsed with a tool based on the Eclipse C Development Toolkit (CDT)[3], in order to add to the meta-model instance information about the application source code structure, such as the functions and their parameters, the call graph, the local and global variables.

The user can manually annotate the source code to select the functions (or parts of their code bodies) and variables that constitute the assets that must be protected, and the related security requirements, such as confidentiality and integrity. These annotations are automatically parsed by the tool and to fill in the meta-model instance.

Then, an automatic attack discovery tool [2, 3], written in Java, uses the information added to the instance by the parser in order to infer the possible attacks able to

[3] `https://www.eclipse.org/cdt/`

endanger the user-defined security requirements of the application's assets. The attacks are inferred via an automated reasoning based on a knowledge base of Prolog rules, obtaining for each security requirement of each asset the set of attack paths able to endanger that requirement.

For each attack path, a set of inference rules is used to the find the protections that can be applied to the endangered asset in order to block the attack. Also, for each proposed application of a protection to an asset, the software metrics of the latter after deploying the protection is estimated using a set of neural networks [4]. In this way, the user may evaluate the protection effectiveness in defending the asset, e.g., by using the potency proposed by Collberg [5].

Then, the user can select some of the proposed protections and combine them into a single solution. Also, a tool can automatically refine the latter by adding additional protections on code areas not sensitive from a security point of view but nonetheless good candidates for undergoing protective transformations in order to hide the application's assets [6]. Otherwise the difference in structure between transformed and non-transformed code fragments would reveal the location of the assets in the whole program to an attacker. The additional protections are inferred by automatically generating and solving (using the IBM ILOG CPLEX[4] solver) a mixed-integer linear problem, in order to maximize the amount of code that must be analysed by an attacker to find the application's assets [6].

In the following sections, we will show how running this work flow on Sumatra is able to infer all the information needed to protect the application. For the sake of brevity, we are going to concentrate on one specific asset, the function compare1 in the sumatra.c source code file, containing one of Sumatra's core algorithms, specifically the one responsible for the comparison between all the DNA sequences in a single dataset. As already said before, we want to preserve the confidentiality and integrity of such asset.

It is worth noting that the ontology reports full information about all the assets we have annotated in the Sumatra application.

4. The application meta-model

In this section, we describe the application's instantiation of the application meta-model of Section 3.2 of the meta-model paper [1]. In this and later sections, numbered instances are taken directly from the ontology file mentioned in Section 2.

The automatic parser is able to automatically add the information about the source code structure of the application in the meta-model instance. We have manually annotated the assets listed in Table 1: such annotations are automatically parsed, resulting in instances of the Asset class. Classes instantiated by the parser are reported in Table 2.

Looking at the compare1 function in sumatra.c, since we have annotated its body as an asset in the source code, from the parsing we obtain in the ontology file:

[4] https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer

Class name	File	ApplicationPart	Asset	Datum	Code	Call	DatumItem
# instances	12	1159	25	909	225	708	1551

Table 2: Classes automatically instantiated by the parser.

DatumType	parameter	genericVariable	integerArrayDatum	integerDatum
# Datum instances	6	9	2	6

Table 3: Datum instances representing parameters and local variables of compare1.

1. a File instance, with the source path of sumatra.c as an attribute;
2. an ApplicationPart instance named applicationPart.714, with the function name as an attribute;
3. an Asset instance named applicationPart.738, contained in applicationPart.714, with two hasRequirement relationships with respectively the confidentiality and integrity elements of the SecurityRequirement enumeration.

Furthermore, applicationPart.714 contains Datum instances representing its parameters (from applicationPart.715 to applicationPart.720) and local variables (from applicationPart.721 to applicationPart.737). Information about variables' type is modelled using an hasType association with the appropriate DatumType enumeration element, as shown in Table 3.

The meta-model represents functions calls, such as the one made by the main function in sumatra.c to compare1 in the same file. The applicationPart.946 instance representing the main function has a hasCall association with the call.2930 instance of the Call class. The latter has a hasCallee association with applicationPart.714 representing compare1. Furthermore, we can look at the first parameter of this call: the main function passes to compare1 as first parameter the variable named db1, represented in the ontology with the applicationPart.959 instance, obviously contained in applicationPart.946. Consequently, call.2930 has a startsWith association with the DatumItem instance datumItem.2930.0, which in turn has a refersTo association with the applicationPart.959 instance representing the passed db1 variable.

5. The attack meta-model

In this section, we describe the instantiation of the attack meta-model of Section 3.4 of the meta-model paper [1] for the Sumatra application.

By running the automatic attack discovery tool, we are able to find the possible attacks against the security requirements of the manually annotated assets. The attacks are modelled as AttackPath instances, each made of a sequence of AttackStep instances. Each of the latter is related to a specific AttackStepType, which is in turn related to the minimum AttackerExpertise needed by the attacker to perform an attack of such type.

AttackStepType	Implementing AttackToolType	Required AttackerExpertise
dynamicStructureAndDataAnalysis	debugger	geek
dynamicTampering	debugger	geek
staticStructuralCodeAndDataRecovery	disassembler, procedureMatchingTool, decompiler	amateur
staticTampering	staticTamperingTool	amateur

Table 4: Instances of class AttackStepType in the ontology.

AttackToolType	# AttackTool instances	Examples of AttackTool	Minimum required AttackerExpertise
debugger	7	gdb, ollyDbg	amateur
decompiler	3	boomerang, smartDec	geek
disassembler	7	hopperv3, viviset	amateur
procedureMatchingTool	4	peiD, unstrip	amateur
staticTamperingTool	5	peTools, metasm	amateur

Table 5: Instances of class AttackTool in the ontology.

Furthermore, an AttackStepType can be performed by an attacker by using an AttackTool of the appropriate AttackToolType, if the attacker has the required AttackerExpertise. For example, even a non-professional attacker may be able to attach a debugger to the application and do some dynamic tampering, such as modifying the value of a variable at runtime: therefore, the AttackStepType dynamicTampering has a requireExpertise relationship with the geek element of the AttackerExpertise enumeration. The dynamicTampering AttackStepType is related with the debugger AttackToolType: however, some debugger are really easy to use while some requires a great expertise from the attacker. For example, the armDSTREAM AttackerTool has a requireExpertise relationship with the expert AttackerExpertise, since Arm DSTREAM[5] is an expensive, professional hardware debugger for Arm System-on-Chips that can be very difficult to use for a non-professional attacker. Attack step types in the ontology, with the implementing attack tool types are summarized in Table 4, while information about attack tools in the ontology is available in Table 5.

We have run the automatic attack discovery tool on Sumatra, in order to automatically instantiate all the classes needed to represent the possible attacks against the manually annotated assets in Sumatra source code. General statistics about the instances of the aforementioned classes are available in Table 6.

For example, an attacker may breach the integrity of the compare1 asset by executing the application with a debugger attached and running a comparison between DNA sequences in the same dataset in order to run the compare1, in order to locate and change the latter using the debugger. In the ontology, this attack is modelled by the attackPath.18 instance of the AttackPath class. Such instance has a startsWith with the AttackStepItem attackStepItem18.0, which in turn has a refersTo association with attackStep.23,

[5]https://developer.arm.com/products/software-development-tools/debug-probes-and-adapters/dstream

Class	AttackPath	AttackStepItem	AttackStep	AttackTarget
# instances	162	353	150	50

Table 6: Instances of automatically instantiated classes in the ontology representing attacks to the application.

representing the first attack step of the path, i.e. the execution of the compare1 function. This AttackStep instance has a hasTarget relationship with the attackTarget.4 AttackTarget instance of the AttackTarget class that has a threatens relationship with the asset applicationPart.738. The same instance has no affects relationships with any SecurityRequirement instance, since executing the function does not breach any security requirement and is only a preliminary step for the attack. Furthermore, the attackStepItem18.0 has a isFollowedBy relationship with attackStepItem18.1, the second step in the attack, which in has a isFollowedBy relationship with attackStepItem18.2, the last step in the attack. Taking a closer look to the final step, we may see as the relative AttackStepItem instance has no isFollowedBy association since it is the last step in the attack path; also, it has a refersTo association with the attackStep.25, representing the alteration of the compare1 function logic by means of a debugger. Therefore, the attackStep.25 has a hasTarget relationship with the attackTarget.5, which in turn has a threatens relationship with the asset applicationPart.738 and a affects relationship with the integrity SecurityRequirement element. Furthermore, the attackStep.25 is related to its type with a hasType relationship with the dynamicTampering element of the AttackStepType enumeration. From the latter, generic information about the attack (possible attack tool used, required attacker expertise, potential mitigations of the attack) can be taken into account when analysing a specific attack step. For example, we can infer that attackStep.25 can be executed even by a non-professional attacker, since the attack step is related to its type dynamicTampering which in turn has a requiresExpertise association with the geek AttackerExpertise.

6. The protection meta-model

In this section, we describe the instantiation of the protection meta-model of Section 3.3 of the meta-model paper [1] for the Sumatra application.

Information about available protections is independent from the analysed application. The Protection class instances describe general protection types, while application of such protections with specific tools are modelled by using the ProtectionInstance class. More precisely, ProtectionInstance instances describe a protection applied by means of the tool X executed with a precise set of tool-specific parameters that drive its deployments. Available protections in the ontology are summarized in Table 7. Furthermore, protections are able to mitigate various AttackStepType instances with different efficacy levels, expressed with Level enumeration. This concept is expressed in the meta-model by using the Mitigation class, and instances of the latter in the ontology are reported in Table 8.

If we restrict the usage of AppliedProtectionInstance objects related to Protection instances with a medium or high Level against at least one of the AttackStep individuals

Protection	# ProtectionInstance
antiDebugging	1
binaryCodeControlFlowObfuscation	9
callStackChecks	1
codeMobility	1
dataObfuscation	3
staticRemoteAttestation	1
whiteBoxCryptography	1

Table 7: Instances of class ProtectionInstance in the ontology.

	dynamicStructure-AndDataAnalysis	dynamic-Tampering	staticStructural-CodeAndDataRecovery	static-Tampering
antiDebugging	medium	high	medium	medium
binaryCodeControlFlowObfuscation			high	medium
callStackChecks		low		
codeMobility	low		low	
dataObfuscation	medium		medium	medium
staticRemoteAttestation		high	low	high
whiteBoxCryptography	medium		medium	

Table 8: Instances of class Mitigation in the ontology, with related Level.

found with the automatic attack discovery tool, we obtain 299 instances of the AppliedProtectionInstance class. The protection discovery tool found at least one AppliedProtectionInstance able to harden each AttackPath, i.e., able to block or make harder at least one AttackStep constituting the AttackPath. For example, to mitigate the aforementioned attackPath.18, the tool suggests two protections that can be applied to the asset compare1, anti-debugging (appliedProtectionInstance.47) and/or static remote attestation (appliedProtectionInstance.64), since the relative Protection are both related to a Mitigation with a high Level against the dynamicTampering AttackStepType, which is the type of the last AttackStep in the examined attack path.

We have then manually combined a selection of the suggested AppliedProtectionInstance into a Solution instance (solution.1 in the ontology). Based on our experience, the protection in that solution are able to safeguard the security requirements of the application's asset effectively. Note that the model is able to represent different protection applied to the same asset: for example, in our solution the compare1.r17 asset is protected with 3 different protections: static remote attestation (appliedProtectionInstance.104), anti-debugging (appliedProtectionInstance.47) and code mobility (appliedProtectionInstance.102).

The Solution instance is described in Table 9, and consists of 27 AppliedProtectionInstance (one per asset). Furthermore, we automatically refined this solution with the asset hiding tool: the resulting solution (solution.2 in the ontology), reported in Table 10, contains the 27 AppliedProtectionInstance of the input solution, and also additional 45 AppliedProtectionInstance on non-assets application parts.

ProtectionInstance name	Asset name	# mitigated AttackPath
Anti-Debugging	cleanDB.r9	8
Code Mobility	printResults.r16	2
Anti-Debugging	printOnlySeqFromFastaSeqPtr.r20	8
Code Mobility	seq_readAllSeq2.r6	2
Anti-Debugging	calculateMaxAndMinLenDB.r13	6
Anti-Debugging	reverseSortSeqsWithCounts.r15	4
Anti-Debugging	seq_printSeqs.r8	2
Anti-Debugging	seq_fillSeqOnlyATGC.r4	4
Anti-Debugging	seq_readNextFromFilebyLine.r1	2
Anti-Debugging	addCounts.r10	2
Anti-Debugging	sortSeqsWithCounts.r14	4
Anti-Debugging	printOnlySeqFromChar.r21	2
Binary Obfuscation	main.r19	2
Anti-Debugging	seq_getNext.r0	4
Anti-Debugging	calculateMaxAndMinLen.r12	4
Code Mobility	compare2.r18	2
Static Remote Attestation	compare1.r17	3
Anti-Debugging	compare1.r17	6
Code Mobility	compare1.r17	2
Anti-Debugging	printHeaderAndSeqFromFastaSeqPtr.r24	2
Code Mobility	seq_fillDigitSeq.r5	2
Anti-Debugging	printOnlyHeaderFromTable.r23	2
Anti-Debugging	seq_fillSeq.r3	4
Anti-Debugging	seq_findSeqByAccId.r7	2
Code Mobility	seq_fillHeader.r2	2
Anti-Debugging	uniqSeqsVector.r11	6
Anti-Debugging	printOnlyHeaderFromFastaSeqPtr.r22	2

Table 9: AppliedProtectionInstance constituting our manually generated solution.

ProtectionInstance name	# original Applied-ProtectionInstance	# additional Applied-ProtectionInstance	# Applied-ProtectionInstance
Anti-Debugging	19	20	39
Binary Obfuscation	1	25	26
Code Mobility	6	0	6
Remote Attestation	1	0	1
Total	27	45	72

Table 10: AppliedProtectionInstance constituting our manually generated solution.

7. Conclusions

This data in brief paper has illustrated, with precise examples from the risk analysis and mitigation of a single asset in the Sumatra application, the descriptive abilities of the meta-model developed to represent information about identifying threats and protecting assets in software applications.

Acknowledgements

This research is supported by the European Union Seventh Framework Programme (FP7/2007-2013), project ASPIRE (Advanced Software Protection: Integration, Research, and Exploitation), under grant agreement no. 609734.

References

[1] C. Basile, D. Cavanese, L. Regano, P. Falcarin, B. De Sutter, A meta-model for software protections and reverse engineering attacks, Journal of Systems and SoftwareUnder submission.

[2] C. Basile, D. Canavese, J. D'Annoville, B. De Sutter, F. Valenza, Automatic discovery of software attacks via backward reasoning, in: Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on, 2015, pp. 52–58.

[3] L. Regano, D. Canavese, C. Basile, A. Viticchié, A. Lioy, Towards automatic risk analysis and mitigation of software applications, in: Information Security Theory and Practice, Springer International Publishing, 2016, pp. 120–135.

[4] D. Canavese, L. Regano, C. Basile, A. Viticchié, Estimating software obfuscation potency with artificial neural networks, in: Security and Trust Management, Springer International Publishing, 2017, pp. 193–202.

[5] C. Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations, Tech. rep., Department of Computer Science, The University of Auckland, New Zealand (1997).

[6] L. Regano, D. Canavese, C. Basile, A. Lioy, Towards optimally hiding protected assets in software applications, in: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017, pp. 374–385.