Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security

Marwan Abi-Antoun
Carnegie Mellon University
mabianto@cs.cmu.edu

Daniel Wang
Microsoft Corporation
daniwang@microsoft.com

Peter Torr
Microsoft Corporation
ptorr@microsoft.com

ABSTRACT

Threat modeling is a lightweight approach to reason about application security and uses Data Flow Diagrams (DFDs) with security annotations. We extended Reflexion Models to check the conformance of an as-designed DFD with an approximation of the as-built DFD obtained from the implementation. We also designed a set of properties and an analysis to help novice designers think about security threats such as spoofing, tampering and information disclosure.

Categories and Subject Descriptors: D.2.4 [Software/Program Verification]: Validation

General Terms: Design, Documentation, Security

Keywords: Threat modeling, Data Flow Diagrams.

1. INTRODUCTION

For several years, Microsoft, Boeing and other companies have been using *threat modeling* [6, 7] to find security design flaws during development. Threat modeling looks at a system from an adversary's perspective to anticipate security attacks and is based on the premise that an adversary cannot attack a system without a means of supplying it with data or otherwise interacting with it. Documenting the system's *entry points*, i.e., interfaces it has with the rest of the world, is crucial for identifying possible vulnerabilities.

Threat modeling uses traditional Data Flow Diagrams (DFDs) [8] with security-specific annotations to describe how data enters, leaves and traverses the system. One large project at Microsoft has over 1,400 completed and reviewed threat modeling DFDs, so we needed a semi-automated approach to support and enhance the current threat modeling process, still mostly manual [4].

We defined a semi-formal DFD representation with an explicit mapping to the implementation (Section 2). Our first contribution is checking the conformance between an as-built DFD and an as-designed DFD (Section 3) using extensions to Reflexion Models [5]. Our second contribution is a security analysis at the level of a DFD (Section 4).

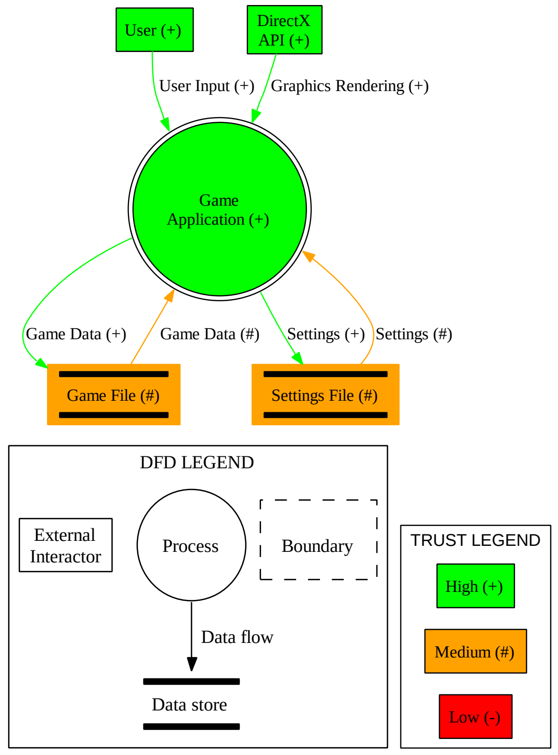

Figure 1: Minesweeper as-designed DFD.

2. DATA FLOW DIAGRAMS

A Data Flow Diagram (DFD) for Minesweeper, a game that ships with Windows ^{TM}, is shown in Figure 1.

We represent a DFD as a runtime view following the Component-and-Connector viewtype [2, pp. 364–365]. A DFD has a fixed set of component types: Process, HighLevelProcess, DataStore, and ExternalInteractor [6].

A Process represents a task in the system that processes data or performs some action based on the data.

A DataStore represents a repository where data is saved or retrieved, but not changed. Examples of data stores include a database, a file, or the Registry — a database of configuration settings in Windows operating systems.

An ExternalInteractor represents an entity that exists outside the system being modeled and which interacts with the system at an entry point: it is either the source or destination of data. Typically, a human who interacts with the system is modeled as an ExternalInteractor.

One connector type, DataFlow, represents data transferred between elements.

Typically, a *Context Diagram* shows the entire functionality of the system represented as a single node. That node is then broken into multiple elements in a *Level-0* diagram. From there, *Level-1*, *Level-2*, etc., diagrams can be constructed. In our approach, we check conformance between two Level-n DFDs, one as-built, and the other as-designed.

3. CHECKING CONFORMANCE

The basic idea to check conformance is to extract the as-built DFD from the implementation and compare it the as-designed DFD using Reflexion Models [5], but with some extensions. In the latter, a *source model* is extracted using a third-party tool from the source code. The user posits the as-designed *high-level model* and then a *mapping* between the source and high-level models. Pushing each interaction described in the source model through the map produces *inferred* edges between high-level model entities. A computation then compares the inferred edges with the edges stated in the high-level model to produce the *reflexion model.*

Match Rules, Function Groups map from the source model to the high-level model.

Match Rules. In Reflexion Models, the mapping between the source model and the high-level model is specified entirely each time. In our approach, a reusable catalog of match rules captures, for a given platform, some of the security domain expertise regarding security-relevant APIs for file access, registry access, network access, command-line access, event log access, etc.

Function Groups. Match Rules are often sufficient to obtain a Level-1 DFD. However, obtaining a Level-2 DFD for an application often requires looking at the internals of the application binaries. When threat modeling, the exposed API of a subsystem is broken up into a set of logically-related functions and each set — rather than each individual function, is modeled as an entry point.

A Function Group logically clusters the services provided by a set of binaries into a DFD element. The earlier Match Rules involve system binaries and are reusable across applications. Function Groups are application- and scenario-specific, so they are added to the DFD model for a given scenario. One function group can involve different binaries as well, since one may want to group in the same logical entry point, a wrapper function defined in a binary that simply calls another function in another binary.

Even for an application as simple as Minesweeper, the conformance check revealed several noteworthy absences. The as-designed DFD (Figure 1) failed to mention two entry points, the Registry and the Game Explorer (See Figure 2). The Game Explorer enforces parental controls, so it must be included as an entry point in the DFD.

Reflexion Models Extensions. The base Reflexion Models technique produced a large number of false positives and required excessive manual input. We needed to extend it make it practical to check existing threat modeling DFDs. Although some of these extensions are specific to DFDs, others are more widely applicable.

First, in Reflexion Models, map entries are considered an *ordered sequence* and not a *set.* Using a set enables a given source model element to map to multiple elements in the high-level model and makes for a more stable map.

Building a map from scratch each time is a laborious process. In one Reflexion Models case study, the map of one product contained more than 1,000 entries [5, p. 372]. We also found ourselves defining similar maps for different systems. Although our match rules and function groups could produce the same flat map as Reflexion Models, they are more structured and separate the shared parts of the map that are reusable across different applications from the application-specific ones.

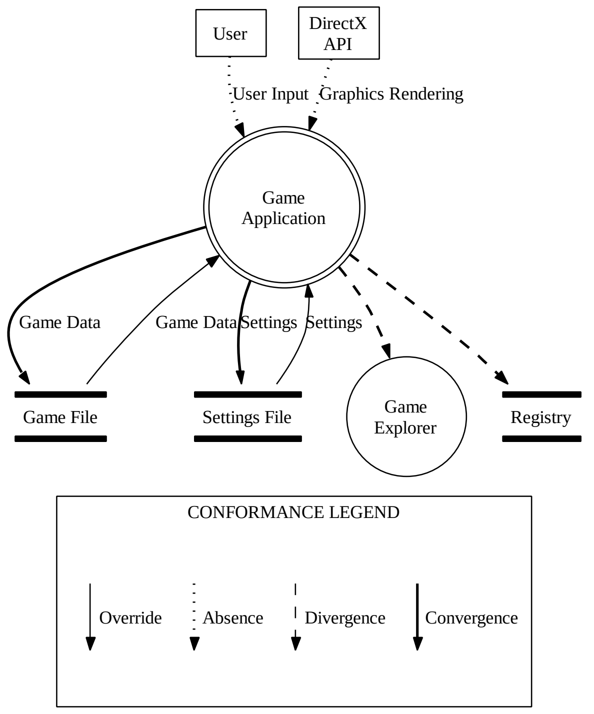

Figure 2: Checking conformance for Minesweeper

Reflexion Models uses eight different inputs and outputs, which makes the interaction needlessly complex: (1) source model; (2) high-level model; (3) mapping; (4) configuration, i.e., node shapes, colors, etc.; (5) exclusions or exceptions; (6) reflexion model; (7) traceability information; and (8) errors or unmapped entries. In later implementations, some of these inputs (e.g., 2,3,4) and outputs were unified.

The computation of the Reflexion Model for DFDs considers the node type defined in the high-level model, and thus requires a more unified representation (Figure 3). BasicEntity, the base component type, stores the conformance finding (discussed below) in the finding property.

Conformance Findings. In Reflexion Models, an edge difference can be: *Convergent*, if it is both in the as-built DFD and in the as-designed DFD; *Divergent*, if it is in the as-built DFD, but not in the as-designed DFD; and *Absent*, if it is in the as-designed DFD, but not in the as-built DFD. We defined the following additional classifications:

- *Excluded:* Self-edges are prohibited in a threat modeling DFD. To reduce the number of false positives, a self-edge that is automatically inferred is automatically marked as *Excluded.*
- *AmbiguousDirection:* in some cases, the analysis cannot determine the directionality of a data flow. To reduce the number of false positives, if an edge exists but its direction cannot be determined, the analysis automatically accepts it as a convergence, but flags it

differently to distinguish it from a true convergence. An edge with an ambiguous direction is shown as a solid edge, as opposed to a bold edge for a true convergence. An example of such a finding is the "Game Data" DataFlow directed from the "Game File" DataStore to the "Game Application" Process in Figure 2.

- *ExternalEntity:* to reduce the number of false positives, we extended Reflexion Models such that the node types in the high-level model can affect the mapping from the source model to the high-level model entities. A DFD does not show a DataFlow between an ExternalInteractor and a DataStore or another ExternalInteractor since those would belong to the DFD of the ExternalInteractor. Such an edge, when inferred, is automatically classified as *ExternalEntity*.

Conformance Exceptions. We generalized Reflexion Models annotations into conformance exceptions. The user can manually override any finding in the Reflexion Model and specify a reason for the override. Each conformance exception documents the original finding, the overridden finding and the reason for the override in the DFD model.

Traceability. In the base Reflexion Models, traceability is discarded between invocations. Our DFD representation represents the traceability information in the high-level model (Figure 3) using *provided* and *required* services.

Each Process, ExternalInteractor and DataStore maintains a list of services it provides. Each DataFlow shows the services involved in the details property, where a connection binds a required service to a provided service.

During conformance checking, traceability information from the source model is added to each DataFlow that is not Absent. For each conformance exception, traceability information consisting of the call information is saved with the conformance exception in the DFD representation. This is used when re-running the analysis: if there is a conformance exception associated with a computed edge, the call information of the computed edge is compared to the previously saved call information. If the analysis detects a discrepancy between the current traceability information and that previously stored with the conformance exception, it flags the exception as suspicious.

Evaluation. An evaluation of the approach on subsystems from production code shows that it can find omitted or outdated information in existing DFDs [1].

4. SECURITY ANALYSIS

High-quality DFDs are often annotated with security relevant information. We formalized a core set of security properties in the DFD representation. Many of these properties are enumerations with pre-defined values and a default value of Unknown. The complete DFD representation with the conformance findings, traceability to code and security properties is shown in Figure 3.

For instance, all DFD elements have a trustLevel property. In addition to the common properties, there are type-specific properties. Some of the possible DataStore-specific properties include readProtection and secrecyMethod.

If no meaningful value for a property is specified, and if providing that additional information can enable additional checks, the analysis requests more information from the user, e.g., by suggesting entering a value for the trustLevel of a DataFlow in relation to its source (other than Unknown).

By definition, the trustLevel of a Process — i.e., code

- **Model**
  - settings: Settings
  - dfd: List<BasicEntity>
- **Settings**
  - binaries: List<String>
  - matchrules: List<MatchRule>
  - functiongroups: List<FunctionGroup>
  - exceptions: List<ConformanceException>
- **MatchRule**
  - name: String // *e.g.,* "Registry"
  - type: ShapeType // *e.g.,* DataStore
  - binary: String // *e.g.,*"advapi32.dll"
  - match: String // *e.g.,* "Reg.+" or "Init"
- **FunctionGroup**
  - name: String // *e.g.,* "Contact APIs"
  - type: ShapeType // *e.g.,* Process
  - services: List<Service>
- **Service**
  - binary: String // *e.g.,* "`app.dll`"
  - function: String // *e.g.,* "`Initialize`"
- **ConformanceException**
  - source: BasicEntity
  - destination: BasicEntity
  - details: List<Connection>
  - finding: Finding
  - reason: String
- **Connection**
  - provided: Service
  - required: Service
- **Finding**: Absent | Convergent | Divergent | Excluded | Overridden | AmbiguousDirection | ExternalEntity | Unknown
- **ShapeType**: Process | DataStore | DataFlow . . .
- **BasicEntity**
  - name: String
  - shapeType: ShapeType
  - services: List<Service>
  - trustLevel: TrustLevel
  - howFound: HowFound
  - owner: Owner
  - finding: Finding
- **Process** extends BasicEntity
  - inputTrustLevel: TrustLevel
  - performsAuthentication: boolean
  - authenticationMethod: Authentication
  - performsAuthorization: boolean
  - authorizationMethod: Authorization
  - performsValidation: boolean
  - validationMethod: Validation
- **DataFlow** extends BasicEntity
  - source: BasicEntity
  - destination: BasicEntity
  - secrecyMethod: Secrecy
  - integrityMethod: Integrity
  - details: List<Connection>
- **DataStore** extends BasicEntity
  - readProtection: Protection
  - writeProtection: Protection
  - secrecyMethod: Secrecy
  - integrityMethod: Integrity
- **ExternalInteractor** extends BasicEntity
- **TrustLevel**: None | High | Medium | Low | Unknown
- **HowFound**: HardCoded | . . . | Mixed | Unknown
- **Owner**: ThisComponent | CompanyTeam | Anybody | . . . | Mixed | Unknown
- **Protection**: None | SystemACLs | . . . | Other
- **Secrecy**: None | Encryption | . . . | Other
- **Integrity**: None | DigitalSignature | . . . | Other
- **Authentication**: None | Windows | . . . | Other
- **Authorization**: None | RoleBased | . . . | Other
- **Validation**: None | ManualParsing | . . . | Other

Figure 3: Extended DFD representation.

that is shipped part of the application, must be High. If that is not the case, then that element must be represented as an ExternalInteractor. A DataStore, ExternalInteractor or a DataFlow can have any trustLevel. Since an element's trustLevel controls several security checks, it is also represented graphically (See Figure 1).

The analysis looks for security flaws such as Spoofing, Tampering, Repudation, Information Disclosure, Denial of Service and Elevation of Privilege (STRIDE) [3, pp. 83–87].

For each of the rules discussed below, the analysis works as follows (we illustrate it with the tampering rule **T1**):

- **Analyze threats:** an attacker tampers with the contents of a DataStore whose trustLevel is High;
- **Analyze mitigations:** if a readProtection and writeProtection are SystemACLs, assume that the threat of tampering is reduced;
- **Suggest remedies**: if readProtection and writeProtection are None, suggest the remedy in the rule: "use Access Control Lists (ACLs)". A remedy is often just an advice for the modeler. Unless the remedy results in changing the values of some properties, the tool cannot always check whether they are performed.

The rules, organized by category, include:

Spoofing. An attacker pretends to be someone else.

- **S1.** Threat: If a DataFlow's trustLevel is higher than the trustLevel of the DataFlow's source, the source can potentially spoof the trusted data;
  Remedy: ensure that the flow is not treated as more trusted than the source entity.
- **S2.** Threat: An ExternalInteractor with a trustLevel other than None can be easily spoofed;
  Remedy: ensure that strong authentication and authorization constraints are in place;
  Mitigation: if performsAuthentication and performsAuthorization are set to true, the methods of authentication and authorization must be set using authenticationMethod and authorizationMethod.
- **S3.** Threat: If howFound property is set to Unknown, the entity has no defined mechanism for being located;
  Remedy: set the howFound property.
- **S4.** Threat: If howFound is set to HardCoded, the location of entity is hard-coded into the system binaries, so it cannot be spoofed.
- **S5.** Threat: If howFound is set to Pointer, the location of this entity is pointed to by another entity;
  Remedy: ensure that the referring entity cannot spoof the name or location of this entity, or cause the system to access an unexpected entity.
- **S6.** Threat: If howFound is set to Mixed, the entity has some hard-coded and some dynamic entities;
  Remedy: include a more detailed diagram that breaks the entity up into individual parts.

Tampering. Data is changed in transit or at rest.

- **T1.** Threat: If the trustLevel of a DataStore is other than None, it is possible for the contents to be tampered with or read by an attacker;
  Remedy: add Access Control Lists (ACLs) to the DataStore or take other precautions;
  Mitigation: readProtection and writeProtection are set to values other than Unknown or None.

Information Disclosure. An attacker steals data while in transit or at rest.

- **I1.** Threat: If the trustLevel of a DataFlow's source is higher than that of its destination, information disclosure is possible;
  Remedy: ensure that no sensitive information is leaked in this flow.
- **I2.** Threat: If the trustLevel of a DataFlow's source has a value that is lower than the trustLevel of the destination, look for potential flaws;
  Remedy: ensure that the destination does not implicitly trust the input as it could be tainted.

Denial of Service. An attacker interrupts the legitimate operation of a system. Such a threat may arise if messages are not validated before use (e.g., by stripping prohibited escape characters), thus allowing a rogue client to crash the system and cause a denial of service for other valid clients.

- **D1.** Threat: An ExternalInteractor with a trustLevel other than High can launch a denial of service attack.

Ownership. To avoid security flaws that arise when teams make different security assumptions about subsystems that may interact, each element is assigned an owner:

- **O1.** Threat: An ExternalInteractor's owner is set to ThisComponent;
  Remedy: mark the entity's owner as being external or convert the entity into a process or other type.
- **O2.** Threat: If an entity's owner is marked as Anybody, its trustLevel must be None since this code can be written by anyone and must be untrusted.
  Remedy: either update the entity's owner or change its trustLevel.
- **O3.** Threat: An entity's owner is CompanyTeam.
  Remedy: trade threat models with the other team so each team is aware of the other's assumptions.
- **O4.** Threat: An entity's owner is Mixed.
  Remedy: expand the entity to have a single owner in a more detailed diagram.

Checking DFDs under development found both ailed sanity checks to missing critical security information.

5. CONCLUSION

The Reflexion Models technique is used to check the conformance between an as-built DFD and an as-designed DFD. The check yields valuable traceability information that can be used by other code quality tools. Finally, extending the DFD representation with security properties enables an analysis of early DFDs produced by developers with limited threat modeling experience.

6. REFERENCES

[1] M. Abi-Antoun, D. Wang, and P. Torr. Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security. Technical Report CMU-ISRI-06-124, Carnegie Mellon University, 2006.

[2] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architecture: View and Beyond.* Addison-Wesley, 2003.

[3] M. Howard and D. LeBlanc. *Writing Secure Code.* Microsoft Press, 2nd edition, 2003.

[4] M. Howard and S. Lipner. *The Security Development Lifecycle.* Microsoft Press, 2006.

[5] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software Reflexion Models: Bridging the Gap between Design and Implementation. *IEEE Trans. Softw. Eng.*, 27(4), 2001.

[6] F. Swiderski and W. Snyder. *Threat Modeling.* Microsoft Press, 2004.

[7] P. Torr. Demystifying the Threat-Modeling Process. *IEEE Security and Privacy*, 03(5):66–70, 2005.

[8] E. Yourdon. *Structured Analysis.* Prentice Hall, 1988.