STRIDE-based security model in Acme

Marwan Abi-Antoun* **Jeffrey M. Barnes‡**

January 2010
CMU-ISR-10-106

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

* Department of Computer Science, Wayne State University
‡ Institute for Software Research, Carnegie Mellon University

Abstract

In earlier work, Abi-Antoun, Wang and Torr defined a model for reasoning about security at the architectural-level, following the STRIDE methodology, which looks for vulnerabilities in the areas of Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege.
The previous security model and checker were implemented using custom code. We now formalize the same model using ADL support for architectural types and properties, and define the checks as logic predicates. Using an ADL gives the benefit of having a declarative model, with less room for error compared to custom code. Moreover, with such a model, power users can more easily add properties and predicates to extend or customize the security analysis.

Marwan Abi-Antoun was supported by Wayne State University.

Jeffrey M. Barnes was supported in part by the Office of Naval Research (ONR), United States Navy, N000140811223 as part of the HSCB project under OSD, by the US Army Research Office (ARO) under grant numbers DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab and DAAD19-01-1-0485, and by the Software Engineering Institute at CMU.

Keywords: Threat modeling, data flow diagrams, architecture-level security analysis, spoofing, tampering, information disclosure, denial of service, architectural description language, Acme

Contents

1 Introduction

For several years, companies such as Boeing, Microsoft, and others have been using *threat modeling* [12, 13] as a lightweight approach to reason about security, to capture and reuse security expertise and to find security design flaws during development [8, 10]. 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 adopts a data flow approach to track the adversary's data and commands as they are processed by the system and determine what assets can be compromised. Any transformation or action on behalf of the data could be susceptible to security threats. For instance, an adversary can jeopardize the application's security by directly invoking some functionality or by supplying the application with malicious data, e.g., by editing a configuration file.

A Data Flow Diagram (DFD) [14] with security-specific annotations is used to describe how data enters, leaves and traverses the system: it shows data sources and destinations, relevant processes that data goes through and trust boundaries in the system. There are many modeling techniques to reason about security more formally [4]. The current threat modeling process uses DFDs [8], perhaps because they are informal, they have a graphical representation, and they are hierarchical, thereby supporting modular decomposition [9]. 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.

DFDs are typically constructed by development teams and later reviewed by security experts. In fact, it is not uncommon for a developer to produce during a security review meeting a DFD of how they think the system works. In many cases, the mere act of having application developers specify the DFD for a subsystem can make several of its security flaws apparent. Although this approach has found a significant number of security design flaws, it suffers from the following two important problems.

DFDs are initially produced by application experts who have little experience in threat modeling, may not be aware of the rules that good threat modeling DFDs should obey or may not apply them in a systematic way. Furthermore, building high-quality DFDs requires security knowledge. Because teams do not receive feedback from security experts until late in the process, these initial DFDs are often of low quality. High-quality DFDs are also annotated with security relevant properties, but knowing what to annotate with what information typically requires threat modeling experience. We implemented an analysis to assist novice DFD designers by warning about possible threats and providing hints for mitigation. Checking DFDs under development found anomalies ranging from failed sanity checks to missing critical security information.

In previous work [2, 3], Abi-Antoun, Wang and Torr presented a definition of a model for reasoning about security at the architectural-level, following the STRIDE methodology commonly used in threat modeling. The previous security model and checker were implemented in a proprietary Microsoft tool using custom code. This paper formalizes the same model, using ADL support for architectural types and properties, and defines the checks as predicates. Using an ADL gives the benefit of having a declarative model, with less room for error compared to custom code. Moreover, with such a model, the tool's users can more easily add properties and predicates, to extend or customize the security analysis.

The rest of this report is organized as follows. Section2 discusses analyzing a DFD for well-formedness and common security flaws. Section 3 discusses some limitations and future work. Finally, Section 4 surveys related work.

2 Security Analysis

High-quality DFDs are often annotated with security relevant properties, but knowing what to annotate with what information typically requires security modeling experience. We implemented an analysis to check syntactic rules and to look for security flaws at the DFD level. The analysis checks the provided information and automatically requests more information where applicable.

2.1 Syntactic Rules

Several structural rules dictate how elements in a DFD may be connected together [6, p. 78]: (1) A Process must have at least one DataFlow entering and one DataFlow exiting; (2) A DataFlow starts or stops at a Process; (3) A DataFlow cannot have the same source and destination; and (4) A DataFlow cannot be bi-directional, since the data in each direction is often different.

2.2 Security Properties

Trust boundaries are one kind of security-specific annotations that are added to a DFD. High-quality DFDs are also 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 1.

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, writeProtection, secrecyMethod and integrityMethod.

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 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.

2.2.1 SyncFamily.acme Acme Family

```
import $AS_GLOBAL_PATH/families/TieredFam.acme;

Family SyncFamily extends TieredFam with {

    analysis isSrcComponent(d1 : SyncCompT, conn : SyncConnT) : boolean =
        connected(conn, d1) and
        exists src : SyncUserT in conn.ROLES | exists put : SyncUseT in d1.PORTS |
            declaresType(src, SyncUserT) and declaresType(put, SyncUseT)
            and attached(src, put);

    analysis isDstComponent(d2 : SyncCompT, conn : SyncConnT) : boolean =
        connected(conn, d2) and
        exists dst : SyncProviderT in conn.ROLES | exists get : SyncProvideT in d2.PORTS |
            declaresType(dst, SyncProviderT) and declaresType(get, SyncProvideT)
            and attached(dst, get);

    analysis pointsTo(d1 : SyncCompT, d2 : SyncCompT) : boolean =
        exists conn : SyncConnT in self.CONNECTORS |
              isSrcComponent(d1, conn) and isDstComponent(d2, conn);

    analysis NoDanglingRoles(r : Role) : boolean =
        size(r.ATTACHEDPORTS) > 0;


    Role Type SyncUserT extends userT with {
        Property syncStatus : int;
```

- **Model**
  - dfd: List<BasicEntity>
- **ShapeType**: Process | DataStore | ExternalInteractor | 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: AuthenticationType
  - performsAuthorization: boolean
  - authorizationMethod: AuthorizationType
  - performsValidation: boolean
  - validationMethod: ValidationType
- **DataFlow** extends BasicEntity
  - source: BasicEntity
  - destination: BasicEntity
  - readProtection: DataAccessProtection
  - writeProtection: DataAccessProtection
  - secrecyMethod: InformationSecrecy
  - integrityMethod: InformationIntegrity
  - details: List<Connection>
- **DataStore** extends BasicEntity
  - readProtection: DataAccessProtection
  - writeProtection: DataAccessProtection
  - secrecyMethod: InformationSecrecy
  - integrityMethod: InformationIntegrity
- **ExternalInteractor** extends BasicEntity
- **TrustLevel**: None | High | Medium | Low | Unknown
- **HowFound**: HardCoded | ... | Mixed | Unknown
- **Owner**: ThisComponent | CompanyTeam | Anybody | ... | Mixed | Unknown
- **DataAccessProtection**: None | SystemACLs | CustomAccessControl | ... | Other | Unknown
- **InformationSecrecy**: None | Encryption | Obfuscation | ... | Other | Unknown
- **InformationIntegrity**: None | DigitalSignature | ... | Other | Unknown
- **AuthenticationType** None | Windows | Mixed | ... | Other | Unknown
- **AuthorizationType** None | RoleBased | Mixed | ... | Other | Unknown
- **ValidationType** None | RegularExpressions | ManualParsing | Mixed | ... | Other | Unknown

Figure 1: Extended DFD representation (Source: [2]).

```
    }
    Component Type SyncCompT extends TierNodeT with {
        Property syncStatus : int;
        Property label : string;
        Property hasDetail : boolean;
        Property detailStatus : int;
        Property traceability : string;
    }
    Connector Type SyncConnT extends CallReturnConnT with {
        Property syncStatus : int;
        Property label : string;
        Property traceability : string;
        Property summary : int;
    }
    Port Type SyncUseT extends useT with {
        Property syncStatus : int;
    }
    Port Type SyncProvideT extends provideT with {
        Property syncStatus : int;
    }
    Role Type SyncProviderT extends providerT with {
        Property syncStatus : int;
    }
}
```

2.2.2 DFD.acme Acme Family

```
Family DFD = {

    analysis NoDanglingRoles(r : Role) : boolean = size(r.ATTACHEDPORTS) > 0;

    analysis isSrcComponentR(processVar : DFDComponent, dataFlowVar : DataFlow) : boolean =
        connected(dataFlowVar, processVar) AND exists src : userDataT in dataFlowVar.ROLES |
            exists put : useDataT in processVar.PORTS | attached(src, put);
    analysis isDstComponentR(processVar : DFDComponent, dataFlowVar : DataFlow) : boolean =
        connected(dataFlowVar, processVar) AND exists dst : providerDataT in dataFlowVar.ROLES |
            exists get : provideDataT in processVar.PORTS | attached(dst, get);

    analysis isTrustLevelOfProcessHigh(componentVar : DFDComponent) : boolean =
        hasValue(componentVar.trustLevel) AND componentVar.trustLevel == High;
    analysis isTrustLevelDataStoreHigh(dataStoreVar : DFDComponent) : boolean =
        hasValue(dataStoreVar.trustLevel) AND !(dataStoreVar.trustLevel == High);
    analysis isLowTrustedConn(dataFlowVar : DataFlow) : boolean =
        hasValue(dataFlowVar.trustLevel) AND dataFlowVar.trustLevel == Low;
    analysis isPartialTrustedConn(dataFlowVar : DataFlow) : boolean =
        hasValue(dataFlowVar.trustLevel) AND dataFlowVar.trustLevel == Medium;
    analysis isHighTrusted(src : DFDComponent,conn : DataFlow) : boolean =
        (isFullTrusted(src) AND isPartialTrustedConn(conn)) OR
        (isFullTrusted(src) AND isLowTrustedConn(conn)) OR
        (isPartialTrusted(src) AND isLowTrustedConn(conn));
    analysis IsTrustDataFlowLowerThanSource(src : DFDComponent,conn : DataFlow) : boolean =
        isSrcComponent(src, conn) AND isHighTrusted(src, conn);
```

```
analysis isTrustLevelOfExternalNone(componentVar : DFDComponent) : boolean =
    hasValue(componentVar.trustLevel) AND !(componentVar.trustLevel == NoneValue);

analysis isHowFoundSet(componentVar : DFDComponent) : boolean =
    hasValue(componentVar.howFound) AND !(componentVar.howFound == UnknownFound);
analysis notFoundFromPointer(componentVar : DFDComponent) : boolean =
    hasValue(componentVar.howFound) AND !(componentVar.howFound == FromPointer);
analysis notFoundFromMixed(componentVar : DFDComponent) : boolean =
    hasValue(componentVar.howFound) AND !(componentVar.howFound == MixedHowFound);
analysis foundIsHardCoded(componentVar : DFDComponent) : boolean =
    hasValue(componentVar.howFound) AND (componentVar.howFound == HardCoded);

analysis trustLevelProtection(dataStoreVar : DFDComponent) : boolean =
    hasValue(dataStoreVar.trustLevel) AND dataStoreVar.trustLevel == NoneValue;
analysis isLowTrusted(componentVar : DFDComponent) : boolean =
    hasValue(componentVar.trustLevel) AND componentVar.trustLevel == Low;
analysis isPartialTrusted(componentVar : DFDComponent) : boolean =
    hasValue(componentVar.trustLevel) AND componentVar.trustLevel == Medium;
analysis isFullTrusted(componentVar : DFDComponent) : boolean =
    hasValue(componentVar.trustLevel) AND componentVar.trustLevel == High;
analysis isMoreTrusted(src : DFDComponent,dest : DFDComponent) : boolean =
    (isFullTrusted(dest) AND isPartialTrusted(src)) OR
    (isFullTrusted(dest) AND isLowTrusted(src)) OR
    (isPartialTrusted(dest) AND isLowTrusted(src));

analysis isSrcComponent(componentVar : DFDComponent,conn : DataFlow) : boolean =
    connected(conn, componentVar) AND exists src : userDataT in conn.ROLES |
        exists put : useDataT in componentVar.PORTS | attached(src, put);
analysis isDstComponent(componentVar : DFDComponent,conn : DataFlow) : boolean =
    connected(conn, componentVar) AND exists dest : providerDataT in conn.ROLES |
        exists get : provideDataT in componentVar.PORTS | attached(dest, get);

analysis dataFlowsFromLowToHigh(src:DFDComponent, dst:DFDComponent, conn:DataFlow) : boolean =
    isSrcComponent(src, conn) AND isDstComponent(dst, conn) AND isMoreTrusted(src, dst);
analysis isTrustLevelHigh(externalEntityVar : DFDComponent) : boolean =
    hasValue(externalEntityVar.trustLevel) AND (externalEntityVar.trustLevel == High);

analysis Remedy1(externalEntityVar : DFDComponent) : boolean =
    hasValue(externalEntityVar.owner) AND !(externalEntityVar.owner == ThisComponent);
analysis Remedy2(dfdComponentVar : DFDComponent) : boolean =
    hasValue(dfdComponentVar.owner) AND !(dfdComponentVar.owner == Anybody);
analysis Remedy3(dfdComponentVar : DFDComponent) : boolean =
    hasValue(dfdComponentVar.owner) AND !(dfdComponentVar.owner == CompanyTeam);
analysis Remedy4(externalEntityVar : DFDComponent) : boolean =
    hasValue(externalEntityVar.owner) AND !(externalEntityVar.owner == MixedOwner);

Property Type AuthenticationTypeT = Enum {None,Windows,Mixed,Other,UnknownAuthen};
Property Type AuthorizationTypeT = Enum {None,RoleBased,Mixed,Other,UnknownAuthor};
Property Type DataAccessProtectionT =
    Enum {None,SystemACLs,CustomAccessControl,Other,UnknownProtection};
Property Type HowFoundT = Enum {HardCoded,FromPointer,MixedHowFound,UnknownFound};
```

```
Property Type InformationIntegrityT = Enum {None,DigitalSignature,Other,UnknownInfoIntegrity};
Property Type InformationSecrecyT = Enum {None,Encryption,Obfuscation,Other,UnknownInfoSec};
Property Type OwnerT =
    Enum {ThisComponent,OtherTeam,Anybody,CompanyTeam,MixedOwner,UnknownOwner};
Property Type PrivacyLevelT = Enum {None,GeneralPII,SensitivePII,Other,UnknownPrivacyLvl};
Property Type TrustLevelT = Enum {Low,Medium,High,NoneValue,UnknownTrustLevel};
Property Type ValidationTypeT =
    Enum {None,RegularExpressions,ManualParsing,Mixed,Other,UnknownValid};

Element Type DFDTypeT = {
    Property trustLevel : TrustLevelT;
    Property privacyLevel : PrivacyLevelT;
    Property howFound : HowFoundT;
    Property owner : OwnerT;
    Property name : string;
}

Role Type providerDataT = {
    rule validDestinationPort = invariant forall p : Port in self.ATTACHEDPORTS |
        declaresType(p, provideDataT);
    rule rule1 = invariant NoDanglingRoles(self);
}
Role Type userDataT = {
    rule validSourcePort = invariant forall p : Port in self.ATTACHEDPORTS |
        declaresType(p, useDataT);
    rule rule0 = invariant NoDanglingRoles(self);
}

Port Type provideDataT;
Port Type useDataT;

Connector Type DataFlow extends DFDTypeT with {
    Role provider : providerDataT = new providerDataT;
    Role user : userDataT = new userDataT;

    Property dataName : string;
    Property flowNumber : int;
    Property readProtection : DataAccessProtectionT;
    Property writeProtection : DataAccessProtectionT;
    Property secrecyMethod : InformationSecrecyT;
    Property integrityMethod : InformationIntegrityT;

    rule exactlyTwoRoles = invariant size(self.ROLES) == 2;
}

Component Type DFDComponent extends DFDTypeT;
Component Type DataStore extends DFDComponent with {
    Property readProtection : DataAccessProtectionT;
    Property writeProtection : DataAccessProtectionT;
    Property secrecyMethod : InformationSecrecyT;
    Property integrityMethod : InformationIntegrityT;
```

```
}
Component Type ExternalEntity extends DFDComponent with {
    Property performsAuthentication : boolean << default = true; >> ;
    Property authenticationMethod : AuthenticationTypeT;
    Property performsAuthorization : boolean << default = true; >> ;
    Property authorizationMethod : AuthorizationTypeT;
}
Component Type Process extends DFDComponent with {
    Property processNumber : int << default = 1; >> ;
    Property TrustLevel : TrustLevelT << default = High; >> ;
    Property performsAuthentication : boolean << default = false; >> ;
    Property authenticationMethod : AuthenticationTypeT;
    Property performsAuthorization : boolean << default = false; >> ;
    Property authorizationMethod : AuthorizationTypeT;
    Property performsValidation : boolean << default = false; >> ;
    Property validationMethod : ValidationTypeT;
    rule processNumberExists = heuristic (self.processNumber > 0);
}

rule DFStartOrEndInProcess = invariant forall dataFlowVar : DataFlow in self.CONNECTORS |
    exists processVar : Process in self.COMPONENTS |
        isSrcComponentR(processVar, dataFlowVar) OR isDstComponentR(processVar, dataFlowVar);
rule NoSelfLoops = invariant forall CS : Component in self.COMPONENTS |
    forall CD : Component in self.COMPONENTS |
        connected(CS, CD) -> CS != CD;

rule TrustLevelOfProcess = heuristic forall componentVar : DFDComponent in self.COMPONENTS |
    declaresType(componentVar, Process) -> isTrustLevelOfProcessHigh(componentVar)
    << label : string = "Must be represented as an ExternalInteractor"; >> ;
rule TrustLevelDataStoreHigh = heuristic forall componentVar : DFDComponent in self.COMPONENTS |
    declaresType(componentVar, DataStore) -> isTrustLevelDataStoreHigh(componentVar)
    << label : string = "Use access control lists"; >> ;
rule TrustDataFlowLowerThanSource = heuristic (exists src : DFDComponent in self.COMPONENTS |
    exists conn : DataFlow in self.CONNECTORS |
        IsTrustDataFlowLowerThanSource(src, conn))
    << label : string = "Ensure that the flow is not treated as more trusted than the source entity"; >> ;
rule TrustLevelOfExternalNone = heuristic forall componentVar : DFDComponent in self.COMPONENTS |
    declaresType(componentVar, ExternalEntity) -> isTrustLevelOfExternalNone(componentVar)
    << label : string = "Ensure that strong authentication and authorization constraints are in place"; >> ;

rule SetHowFound = invariant forall componentVar : DFDComponent in self.COMPONENTS |
    declaresType(componentVar, DFDComponent) -> isHowFoundSet(componentVar);
rule NoComponentsFoundFromPointer = heuristic
    forall componentVar : DFDComponent in self.COMPONENTS |
        declaresType(componentVar, DFDComponent) -> notFoundFromPointer(componentVar)
    << label : string = "Ensure that the referring entity cannot spoof the name
        or location of the entity that is pointed to by another entity"; >> ;
rule NoComponentsFoundFromMixed = heuristic
    exists componentVar : DFDComponent in self.COMPONENTS |
        declaresType(componentVar, DFDComponent) -> notFoundFromMixed(componentVar)
    << label : string = "Include a more detailed diagram that breaks the entity up into individual parts"; >> ;
```

```
    rule SetComponentsFoundToHardCoded = heuristic
        forall componentVar : DFDComponent in self.COMPONENTS |
            declaresType(componentVar, DFDComponent) -> foundIsHardCoded(componentVar)
        << label : string = "The location of entity is hard-coded into
            the system binaries, so it cannot be spoofed"; >> ;

    rule TrustLevelDataStore = heuristic forall componentVar : DFDComponent in self.COMPONENTS |
        declaresType(componentVar, DataStore) -> trustLevelProtection(componentVar)
        << label : string = "Add access control lists to the data store or take other precautions"; >> ;

    rule NoDataFlowsFromHighToLow = heuristic (exists src : DFDComponent in self.COMPONENTS |
        exists dest : DFDComponent in self.COMPONENTS |
            exists conn : DataFlow in self.CONNECTORS |
                dataFlowsFromLowToHigh(dest, src, conn))
        << label : string = "Ensure that no sensitive information is leaked in this flow"; >> ;
    rule NoDataFlowsFromLowToHigh = heuristic !(exists src : DFDComponent in self.COMPONENTS |
        exists dest : DFDComponent in self.COMPONENTS |
            exists conn : DataFlow in self.CONNECTORS |
                dataFlowsFromLowToHigh(src, dest, conn))
        << label : string = "Ensure that the destination does not
            implicitly trust the input as it could be tainted"; >> ;

    rule TrustLevelOfExternalEntityHigh = invariant
        forall componentVar : DFDComponent in self.COMPONENTS |
            declaresType(componentVar, ExternalEntity) -> isTrustLevelHigh(componentVar)
        << label : string = "Trust level other than High can launch a denial-of-service attack"; >> ;

    rule Ownership1 = heuristic forall componentVar : DFDComponent in self.COMPONENTS |
        declaresType(componentVar, ExternalEntity) -> Remedy1(componentVar)
        << label : string = "Mark the entity's owner as being external
            or convert the entity into a process or other type"; >> ;
    rule Ownership2 = heuristic forall componentVar : DFDComponent in self.COMPONENTS |
        declaresType(componentVar, DFDComponent) -> Remedy2(componentVar)
        << label : string = "Either update the entity's owner or change its trustLevel"; >> ;
    rule Ownership3 = heuristic forall componentVar : DFDComponent in self.COMPONENTS |
        declaresType(componentVar, DFDComponent) -> Remedy3(componentVar)
        << label : string = "Trade threat models with the other team
            so each team is aware of the other's assumptions"; >> ;
    rule Ownership4 = heuristic forall componentVar : DFDComponent in self.COMPONENTS |
        declaresType(componentVar, ExternalEntity) -> Remedy4(componentVar)
        << label : string = "Include a more detailed diagram where the entity is
            expanded to show each individual entity with a single owner"; >> ;
}
```

2.3 Analysis Overview

The analysis looks for security flaws in the STRIDE model, which include Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege [6, pp. 83–87] and are based on the principles of information confinement [5].

At a high level, the analysis works as follows, for each of the rules discussed below. 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." A remedy is often just an informational message for the modeler. Unless the remedy requires changing the DFD or its security properties, the tool cannot always check whether it is performed.

This rule is implemented in DFD.acme as follows:

```
rule TrustLevelDataStoreHigh = heuristic forall componentVar : DFDComponent in self.COMPONENTS |
    declaresType(componentVar, DataStore) -> isTrustLevelDataStoreHigh(componentVar)
    << label : string = "Use access control lists"; >> ;
```

This rule checks that no component of type DataStore has a trust level of High and suggests a remedy if there *is* such a component.

We list below some of the other rules we have implemented, organized by category.

2.4 Spoofing

In 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.
  Implementation: We check that each DataFlow has a trustLevel no higher than its source.

```
rule TrustDataFlowLowerThanSource = heuristic (exists src : DFDComponent in self.COMPONENTS
    exists conn : DataFlow in self.CONNECTORS |
        IsTrustDataFlowLowerThanSource(src, conn))
    << label : string = "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.
  Implementation: We check whether any ExternalEntity has a trustLevel other than None.

```
rule TrustLevelOfExternalNone = heuristic
    forall componentVar : DFDComponent in self.COMPONENTS |
        declaresType(componentVar, ExternalEntity) -> isTrustLevelOfExternalNone(componentVar)
    << label : string = "Ensure that strong authentication
        and authorization constraints are in place"; >> ;
```

- **S3.** Threat: If howFound property is set to Unknown, the entity has no defined mechanism for being located.
  Remedy: Set the howFound property.
  Implementation: We check that howFound is set for every DFDComponent.

```
rule SetHowFound = invariant forall componentVar : DFDComponent in self.COMPONENTS |
    declaresType(componentVar, DFDComponent) -> isHowFoundSet(componentVar);
```

- **S4.** Threat: If howFound is set to HardCoded, the location of entity is hard-coded into the system binaries, so it cannot be spoofed.
  Implementation: We check that every DFDComponent has a howFound value of HardCoded.

```
rule SetComponentsFoundToHardCoded = heuristic
    forall componentVar : DFDComponent in self.COMPONENTS |
        declaresType(componentVar, DFDComponent) -> foundIsHardCoded(componentVar)
```

<< **label** : **string** = "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.
Implementation: We check that no DFDComponent has a howFound value of Pointer.

```
rule NoComponentsFoundFromPointer = heuristic
    forall componentVar : DFDComponent in self.COMPONENTS |
        declaresType(componentVar, DFDComponent) -> notFoundFromPointer(componentVar)
    << label : string = "Ensure that the referring entity cannot spoof the name
        or location of the entity that is pointed to by another 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.
Implementation: We check that no DFDComponent has a howFound value of MixedHowFound.

```
analysis notFoundFromMixed(componentVar : DFDComponent) : boolean =
    hasValue(componentVar.howFound) AND !(componentVar.howFound == MixedHowFound);
```

2.5 Tampering

In 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 to the DataStore or take other precautions.
Mitigation: readProtection and writeProtection are set to values other than Unknown or None.
Implementation: Described above.

2.6 Information Disclosure

In nformation 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.
Implementation: We ensure that there is no DataFlow connector whose source is more trusted than its destination.

```
rule NoDataFlowsFromHighToLow = heuristic (exists src : DFDComponent in self.COMPONENTS |
    exists dest : DFDComponent in self.COMPONENTS |
        exists conn : DataFlow in self.CONNECTORS |
            dataFlowsFromLowToHigh(dest, src, conn))
    << label : string = "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.
Implementation: We ensure that there is no DataFlow connector whose source is less trusted than its destination.

```
rule NoDataFlowsFromLowToHigh = heuristic !(exists src : DFDComponent in self.COMPONENTS |
    exists dest : DFDComponent in self.COMPONENTS |
        exists conn : DataFlow in self.CONNECTORS |
            dataFlowsFromLowToHigh(src, dest, conn))
```

<< **label** : **string** = "Ensure that the destination does not
implicitly trust the input as it could be tainted"; >> ;

2.7 Denial of Service

In *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.
  Implementation: We check that each ExternalEntity has a high trustLevel.

  rule TrustLevelOfExternalEntityHigh = **invariant**
  forall componentVar : DFDComponent **in self.COMPONENTS** |
  declaresType(componentVar, ExternalEntity) -> isTrustLevelHigh(componentVar)
  << **label** : **string** = "Trust level other than High can launch a denial-of-service attack"; >> ;

2.8 Ownership

To avoid security flaws that can arise when different development teams make different security assumptions about subsystems that may interact (e.g., [12, p. 75]), 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.
  Implementation: We check that no ExternalEntity has its owner set to ThisComponent.

  rule Ownership1 = **heuristic forall** componentVar : DFDComponent **in self.COMPONENTS** |
  declaresType(componentVar, ExternalEntity) -> Remedy1(componentVar)
  << **label** : **string** = "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.
  Implementation: We check that no DFDComponent has its owner set to Anybody.

  rule Ownership2 = **heuristic forall** componentVar : DFDComponent **in self.COMPONENTS** |
  declaresType(componentVar, DFDComponent) -> Remedy2(componentVar)
  << **label** : **string** = "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.
  Implementation: We check that no DFDComponent has its owner set to CompanyTeam.

  rule Ownership3 = **heuristic forall** componentVar : DFDComponent **in self.COMPONENTS** |
  declaresType(componentVar, DFDComponent) -> Remedy3(componentVar)
  << **label** : **string** = "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: include a more detailed diagram where the entity is expanded to show each individual entity with a single owner.
  Implementation: We check that no ExternalEntity has its owner set to MixedOwner.

```
rule Ownership4 = heuristic forall componentVar : DFDComponent in self.COMPONENTS |
    declaresType(componentVar, ExternalEntity) -> Remedy4(componentVar)
    << label : string = "Include a more detailed diagram where the entity is
        expanded to show each individual entity with a single owner"; >> ;
```

2.9 Repudiation

In *repudiation*, an attacker performs an action that cannot be traced back to them. However, the analysis currently does not check repudiation since it requires audit trails and there is no easy way to mark that a datum is auditable. A repudiation threat could arise if data that was supposed to be audited was easily tampered with or there were flows that did not include the auditing step.

2.10 Elevation Of Privilege

Similarly, in *elevation of privilege*, an attacker performs actions they are not authorized to perform. Currently, our properties do not track when a *lower-trust* entity or data can influence a *higher-trust* entity that does not perform sufficient validation.

2.11 Summary

The goal of the DFD-level analysis is not to replace a review by security experts. Threat model analysis can only be partially automated since it still requires human creativity and intuition to uncover subtle security flaws that cannot easily be generalized. Nevertheless, the analysis can help inexperienced threat modelers identify many of the small issues by providing suggestions on how to fix them in some cases and encouraging designers to focus on the complex interactions between subsystems. Over time, more properties can be added and additional checks could improve the quality of the analysis.

3 Limitations

There are several limitations to our approach that we plan on addressing in future work.

Information flow. The checks we currently implemented focus on information flow vulnerabilities (Spoofing, Tampering, Information Disclosure). Other checks (Denial of Service, Elevation of Privilege) require tracking state changes in the system.

4 Related Work

STRIDE methodologies. Many references on STRIDE discuss similar checks, albeit in an ad-hoc fashion [7, 12]. We are unaware of a resource that defines element-level properties, or one that defines rules in terms of these elemental properties. These aspects are critical to making an automated analysis work, and are key novel contributions of our work.

Threat Modeling Tools. Microsoft has made publicly available a threat modeling tool [12, 11]. Similarly, the EU-funded CORAS Project [1] proposed its description and tool support for threat modeling. While these tools recently added enhanced architectural-level analyses, neither tool currently relates a security architecture to an implementation. Moreover, the tools implement their analyses using custom code, rather than declarative types and predicates.

5 Conclusion

We formalized an architectural-level security analysis using ADL support for architectural types and properties, and define the checks as logic predicates. Using an ADL gives the benefit of having a declarative

model, with less room for error compared to custom code. Moreover, with such a model, power users can more easily add properties and predicates to extend or customize the security analysis.

Acknowledgements

The idea of re-implementing a STRIDE-based security model [2, 3] using types and predicates in the Acme ADL was inspired by ongoing discussions with David Garlan, Kirti Garg and Bradley Schmerl at Carnegie Mellon University. The authors are grateful to Bradley Schmerl for his help with Acme and AcmeStudio. The authors also thank Raed Almomani for his contributions.

References

[1] CORAS. `http://coras.sourceforge.net`, 2006.

[2] 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, September 2006.

[3] M. Abi-Antoun, D. Wang, and P. Torr. Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security (Short Paper). In *Automated Software Engineering*, pages 393–396, 2007.

[4] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Trans. Comp. Syst.*, 8(1), 1990.

[5] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications ACM*, 19(5):236–243, 1976.

[6] L. Hochstein and M. Lindvall. Diagnosing Architectural Degeneration. In *NASA Goddard Software Engineering Workshop*, pages 137–142, 2003.

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

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

[9] G. T. Leavens, T. Wahls, and A. L. Baker. Formal Semantics for Structured Analysis Style Data Flow Diagram Specification Languages. In *SAC*, 1999.

[10] S. Lipner. The Trustworthy Computing Security Development Lifecycle. In *Annual Computer Security Applications Conference*, pages 2–13, 2004.

[11] Microsoft threat modeling tool. `http://msdn.microsoft.com/en-us/security/sdl-threat-modeling-tool.aspx`, 2007.

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

[13] P. Torr. Demystifying the Threat-Modeling Process. *IEEE Security and Privacy*, 3(5):66–70, 2005.

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