



Università
degli Studi
della Campania
Luigi Vanvitelli

Pitch


Press Space for next page →



MAL

Meta Attack Language

- Design di linguaggi domain-specific;
- Fornisce un formalismo che consente la generazione semi-automatizzata e il calcolo efficiente di "grafi di attacco" ^[1] molto grandi; per esempio, un attaccante può
 1. Compromettere un host
 2. Trovare le credenziali salvate
 3. Utilizzare le credenziali per accedere ad un altro host
- Il risultato finale di una simulazione è quello di stimare il tempo impiegato in ciascun passaggio per eseguire l'attacco.

1. Grafi diretti che rappresentano le dipendenze tra i passaggi che possono essere eseguiti dall'attaccante. 

Modelling only security languages

- **CORAS**: Common Object-oriented Representation of Attack Scenarios
- **secureTROPOS**: Secure Threat and Risk Object-oriented Process
- **SecDSVL**: Secure Domain Specific Vulnerability Language

Questi linguaggi permettono di modellare le proprietà di sicurezza di un sistema, ma non sono in grado di fornire alcuno strumento automatico per analizzare o inferire conclusioni sulla sicurezza del modello stesso.

Formalism for Threat Modeling

- Sia x un'entità del dominio, ad esempio, un'entità potrebbe essere un diamante *cullinanDiamond*, un'altra una cassaforte *antwerpVault*.

- Gli oggetti sono divisi in un set di classi $X = \{X_1, \dots, X_n\}$, e.g.

cullinanDiamond \in *Jewelry*

e

antwerpVault \in *Vault*

- A ciascuna classe è associato un set di step di attacco $A(X_i)$.
 - Denotiamo con $X.A$ lo step di attacco per l'oggetto nella classe X .
 - Esempi di step di attacco possono essere *Jewelry.steal* o *Vault.open*.
- Un link è una tupla di oggetti, ciascuno appartenente ad una classe diversa, e rappresenta una dipendenza tra gli oggetti, e.g.

$\lambda = (x_i, x_j)$

- Ad esempio, *antwerpVault* potrebbe avere un link *containment* con *cullinanDiamond*, che rappresenta il fatto che la cassaforte può contenere il diamante.
- I link sono divisi in set di associazioni $\Lambda = \{\Lambda_1, \dots, \Lambda_n\}$, che collega una classe all'altra, e.g.

$x_i, x_k \in X_m, x_j, x_l \in X_n | \lambda_1 = (x_i, x_j) \in \Lambda, \lambda_2 = (x_k, x_l) \in \Lambda$

- Nelle associazioni, le classi ricoprono dei ruoli $\Psi(X_i, \Lambda)$.
 - Ad esempio, *antwerpVault* potrebbe avere il ruolo *container* nella relazione *containment*.
 - Analogamente, a livello di istanze,

$\psi(x_i, \Lambda) = \{x_j | x_i, x_j \in \lambda \wedge \lambda \in \Lambda\}$

Ad esempio,

antwerpVault.contained = {*cullinanDiamond*, *lesothoPromise*}

- Navigando tra le associazioni, gli step di attacco possono essere connessi tra di loro attraverso degli archi diretti $e \in E$.

$e = (X_i, A_k, X_j, A_l) | X_j = \Psi(X_i, \Lambda)$

- Una connessione tra due step di attacco implica che dal primo step si può passare al secondo, e.g.

$e = (Vault.open, Vault.contained.steal), e \in E$

sta a significare che se si riesce ad aprire la cassaforte, si possono rubare i diamanti contenuti.

Formalism for Threat Modeling (cont.)

- Una classe obbligatoria è il singleton **Attacker**, $\Xi \in X$, contenente un singolo step di attacco $\Xi.\xi$, il quale rappresenta il punto di partenza dell'attacco sul modello del sistema.
- Per ciascuno step di attacco $A(X_i)$, è definito un **tempo di attacco locale** (stimato) $\phi(A) = P(T_{loc}(A) = t)$.
- Per esempio, il tempo di attacco locale per aprire la cassaforte potrebbe essere specificato da una distribuzione Gamma, con media 12 e deviazione standard di 6 ore,

$$\phi(antwerpVault.open) = Gamma(24, 0.5)$$

- Gli step di attacco possono essere di tipo **AND** o **OR**, $t(X.A) \in \{OR, AND\}$
- **OR** significa che l'attaccante può iniziare lo step di attacco se almeno uno dei passaggi precedenti è stato completato.
- **AND** significa che l'attaccante può iniziare lo step di attacco solo se tutti i passaggi precedenti sono stati completati.
- Per esempio, se $t(Vault.open) = AND$ e

$$(myVault.compromise, myVault.open) \in E$$

e

$$(myVaultPhysicalLocation.access, myVault.open) \in E$$

allora, per poter aprire la cassaforte, è necessario che l'attaccante sia riuscito a compromettere il sistema (ottenere la chiave della cassaforte) e ad accedere alla sua posizione fisica.

- La distribuzione di probabilità del tempo totale richiesto per completare l'attacco è definita come

$$\Phi(A) = P(T_{glob}(A) = t)$$

- Oltre a definire gli step di attacco, le classi possono anche prevedere delle **difese** $D(X_i)$, che rappresentano le azioni che possono essere intraprese per prevenire l'attacco. Utilizziamo $X.D$ per indicare la difesa per l'oggetto nella classe X .
- Una difesa può essere **TRUE** o **FALSE**, $s(D) \in \{TRUE, FALSE\}$, che indica se la difesa è attiva o meno.
- Ad esempio, $Vault.timeLocked$ può essere **TRUE** o **FALSE**.
- Una difesa può essere genitore di uno step di attacco, $(X_i.D, X_j.A) \in E$, e.g.

$$(Vault.timeLocked, Vault.open) \in E$$

che significa che se la cassaforte è bloccata per un certo periodo di tempo ($Vault.timeLocked = TRUE$), allora non è possibile aprire la cassaforte.

Global time to compromise

Il calcolo del tempo globale per compromettere il sistema non solo tiene conto dei tempi locali per ciascuno step di attacco, ma anche dell'ordine di esecuzione degli step di attacco stessi.

Ad esempio, un attaccante con scarse capacità di pianificazione potrebbe essere modellato attraverso una "random walk" sul grafo degli step di attacco. In tal caso, la distribuzione di probabilità per la scelta del prossimo step di attacco è uniforme.

Un attaccante esperto, invece, sceglie sempre il **percorso minimo** per raggiungere lo step di attacco successivo.

Il Meta Attack Language

Il modello matematico sopra descritto potrebbe essere codificato con un linguaggio di programmazione, come ad esempio Java, ma è molto più semplice utilizzare un linguaggio di alto livello, come il Meta Attack Language (MAL).

Gli autori hanno realizzato un compilatore che, a partire da una descrizione del modello in MAL, genera del codice in Java, che può essere utilizzato per calcolare il tempo globale per compromettere il sistema.

Classi

Come specificato sopra, le classi sono le entità fondamentali di una specifica MAL. Una classe è definita come segue:

```
class Channel{  
  | transmit  
    -> parties.connect  
}
```

- Il simbolo ``|`` indica che lo step di attacco è di tipo OR; per cui, se almeno uno degli step di attacco genitori è stato completato, allora lo step di attacco può essere iniziato.
- Step di attacco di tipo AND sono indicati con il simbolo ``&``, mentre le difese sono indicate con il simbolo ``#``.
- La freccia ``->`` indica che la compromissione dello step di attacco `transmit`` apre allo step di attacco `parties.connect``.

Ruoli di associazione

``parties`` è un ruolo di associazione, definito in un altro punto della specifica MAL.

```
associations {  
  Machine [parties] 2-* <-- Communication --> * [channels] Channel  
}
```

Come per l'UML, le associazioni terminano con una cardinalità, che indica il numero di istanze di una classe che possono essere associate ad un'altra classe. Nell'esempio, un oggetto della classe ``Channel`` deve essere associato ad almeno 2 istanze della classe ``Machine``.

Entrambi gli estremi dell'associazione hanno dei ruoli, che sono utilizzati per la navigazione. Per cui, ``myChannel.parties`` si riferisce al set di oggetti della classe ``Machine`` associati a ``myChannel``.

Ereditarietà

Per permettere il riuso delle classi, MAL fornisce un meccanismo di ereditarietà, che è simile a quello dei linguaggi di programmazione orientati agli oggetti.

```
abstractClass Machine {  
  | connect  
  -> compromise  
  & compromise  
  -> channels.transmit  
}  
class Hardware extends Machine {  
  
}  
class Software extends Machine {  
  & compromise  
  -> channels.transmit,  
    executors.connect,  
}
```

Nell'esempio, la classe `Machine` è astratta, e non può essere istanziata. Essa viene specializzata nelle classi concrete `Hardware` e `Software`. In particolare, la classe `Software` eredita tutti gli step di attacco e le associazioni. Inoltre, la classe `Software` effettua l'override dello step di attacco `compromise`, aggiungendo un nuovo step di attacco figlio, `executors.connect`.

Il Meta Attack Language

Tempo di compromissione

Alcuni step di attacco possono essere completati senza nessuno sforzo da parte dell'attaccante. Tuttavia, per altri step di attacco, l'attaccante deve effettuare un certo numero di tentativi prima di riuscire. Questo implica del tempo necessario per completare lo step di attacco. Ad esempio, per crackare una password con un attacco a dizionario, potrebbero essere necessarie 18 ore. Per questo motivo, MAL fornisce un meccanismo per specificare il tempo necessario per completare uno step di attacco.

```
class Credentials {  
    & dictionaryCrack [18.0]  
}
```

Certe volte, però, non siamo a conoscenza del tempo esatto necessario per completare uno step di attacco. In questo caso, possiamo far uso di distribuzioni di probabilità.

```
class Credentials {  
    & dictionaryCrack [GammaDistribution(1.5, 15)]  
}
```

Difese

Abbiamo visto che le classi possono prevedere delle difese, che rappresentano le azioni che possono essere intraprese per prevenire l'attacco. Tecnicamente, ogni difesa include una fase di attacco. Se la difesa è falsa, allora, al momento dell'istanza, il passaggio di attacco associato è contrassegnato come compromesso.

Ad esempio, le credenziali possono essere crittate, e quindi non possono essere compromesse con un attacco a dizionario.

```
class Credentials {  
  | access  
    -> compromiseUnencrypted  
& compromiseUnencrypted  
  # encrypted  
    -> compromiseUnencrypted  
}
```

- Se `Credentials.encrypted` è falso, l'attaccante sarà in grado di raggiungere il passo di attacco `compromiseUnencrypted` non appena ha raggiunto `access`.
- Se, invece, `Credentials.encrypted` è vero, allora `compromiseUnencrypted` non sarà raggiunto, poiché la sua compromissione richiede quella di entrambi i genitori.

Esempio

- Classe [astratta]: `Machine``
 - Specializzazioni: `Hardware`` e `Software``
 - Le macchine possono eseguire Software, ad esempio una workstation può eseguire un sistema operativo, che a sua volta può eseguire un'applicazione.
- Due o più macchine possono essere collegate da un `Channel``
 - Ad esempio, un software per browser web può essere collegato a un software server web tramite un canale https.
- Classe: `Credentials``
 - Le istanze possono essere, ad esempio, nomi utente e password o chiavi private.
 - Le credenziali hanno destinazioni (`Machine``), per le quali fungono da autenticazione e possono essere memorizzate sulle macchine.

```
abstractClass Machine {  
  | connect  
  | -> compromise  
  | authenticate  
  | -> compromise  
  & compromise  
  | -> _machineCompromise  
  | _machineCompromise
```