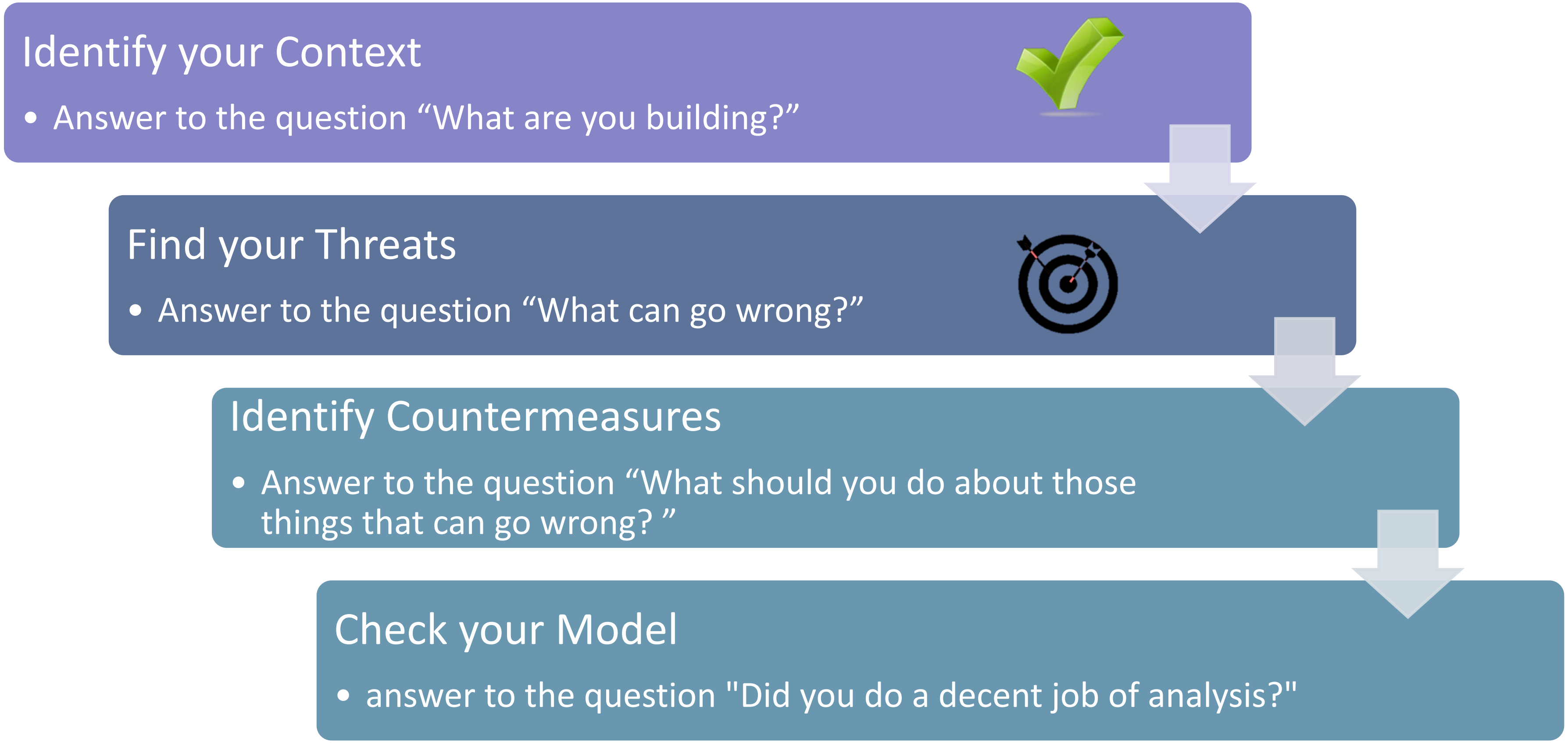
Security Governance
Master of Science in Cyber Security

AA 2023/2024

STRIDE, ATTACK TREES AND ATTACK LIBRARIES

Recap
Basic Steps for Threat Modelling



Identify your Context

- Answer to the question “What are you building?”

Find your Threats

- Answer to the question “What can go wrong?”

Identify Countermeasures

- Answer to the question “What should you do about those things that can go wrong? ”

Check your Model

- answer to the question "Did you do a decent job of analysis?"

STRIDE

The STRIDE approach to threat modelling was invented by Loren Kohnfelder and Praerit Garg in 1999.

It falls mainly in the category of Software-centric Threat Modelling framework

- It was designed to help people developing software to identify the types of attacks that software tends to experience

Using STRIDE can be considered as a Threat elicitation technique

STRIDE

STRIDE classification

THREAT	VIOLATED PROPERTY	THREAT DEFINITION	TYPICAL VICTIMS	EXAMPLES
Spoofing	Authentication	Pretending to be something or someone other than yourself	Processes, external entities	Falsely claiming to be Acme.com, winsock .dll, Barack Obama, a police officer, or the Nigerian Anti-Fraud Group

Spoofing Threats

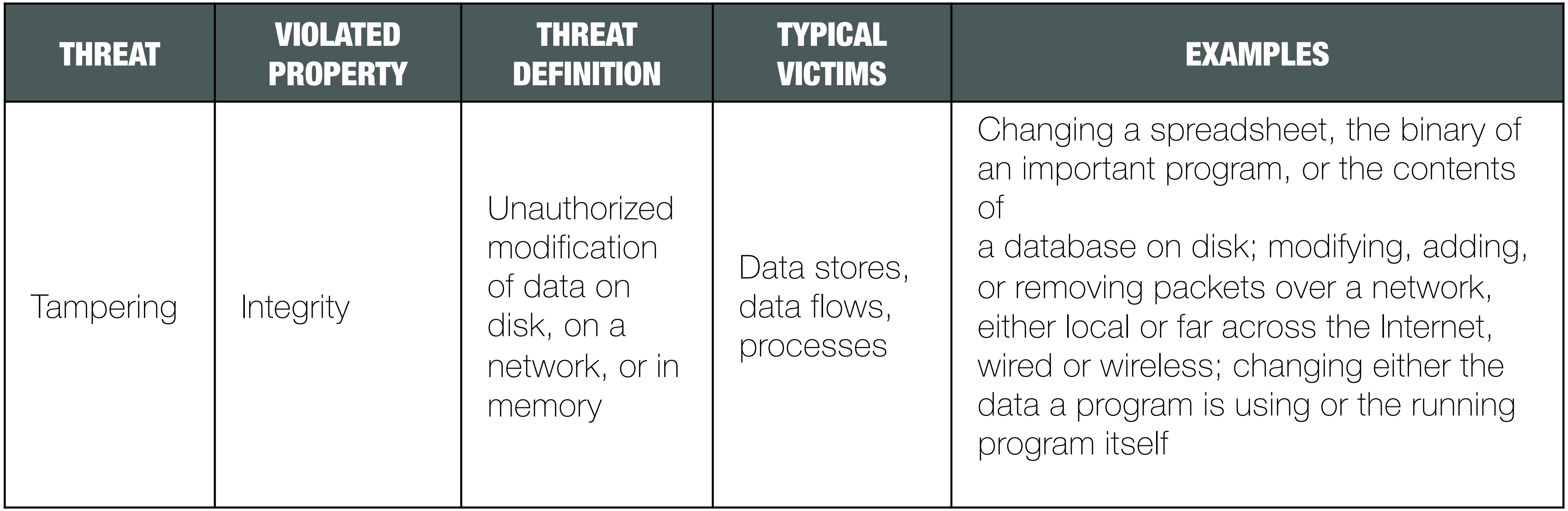
THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Spoofing a process on the same machine	Create a file before the real process	
	Renaming/linking	Creating a Trojan “su” and altering the path
	Renaming	Naming your process “sshd”
Spoofing a File	Create a file in the local directory	This can be a library, executable, or config file
	Creates a link and changes it	From the attacker’s perspective, the change should happen between the link being checked and the link being accessed
	Creates many files in the expected directory	Automation makes it easy to create 10,000 files in /tmp, to fill the space of files called /tmp /”pid.NNNN, or similar

Spoofing Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Spoofing a Machine	ARP spoofing	
	IP spoofing	
	DNS spoofing	Forward or reverse
	DNS compromise	Compromise TLD, registrar or DNS operator
	IP redirection	At the switch or router level
Spoofing a person	Sets e-mail display name	
	Takes over a real account	
Spoofing a role	Declares themselves to be that role	Declares themselves to be that role

STRIDE

STRIDE classification



THREAT	VIOLATED PROPERTY	THREAT DEFINITION	TYPICAL VICTIMS	EXAMPLES
Tampering	Integrity	Unauthorized modification of data on disk, on a network, or in memory	Data stores, data flows, processes	Changing a spreadsheet, the binary of an important program, or the contents of a database on disk; modifying, adding, or removing packets over a network, either local or far across the Internet, wired or wireless; changing either the data a program is using or the running program itself

Tampering Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Tampering with a file	Modifies a file they own and on which you rely	
	Modifies a file you own	
	Modifies a file on a file server that you own	
	Modifies a file on their file server	Loads of fun when you include files from remote domains
	Modifies a file on their file server	Ever notice how much XML includes remote schemas?
	Modifies links or redirects	

Tampering Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Tampering with memory	Modifies your code	Hard to defend against once the attacker is running code as the same user
	Modifies data they've supplied to your API	Pass by value, not by reference when crossing a trust boundary
Tampering with a network	Redirects the flow of data to their machine	Often stage 1 of tampering
	Modifies data flowing over the network	Even easier and more fun when the network is wireless (WiFi, 3G, etc)
	Enhances spoofing attacks	

STRIDE

STRIDE classification

THREAT	VIOLATED PROPERTY	THREAT DEFINITION	TYPICAL VICTIMS	EXAMPLES
Repudiation	Non-Repudiation	• Claiming that you didn’t do something, or were'not responsible. • It can be honest or false • Often apperars at the business level • The key question for system designers is, what evidence do you have?	Processes	Process or system: “I didn’t hit the big red button” or “I didn’t order that Ferrari.” Note that repudiation is somewhat the odd-threat-out here; it transcends the technical nature of the other threats to the business layer.

Repudiation Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Repudiating an action	Claims to have not clicked	Maybe they really did
	Claims to have not received	Receipt can be strange; does mail being downloaded by your phone mean you’ve read it? Did a network proxy pre-fetch images? Did some- one leave a package on the porch?
	Claims to have been a fraud victim	
	Uses someone else’s account	
	Uses someone else’s payment instrument without authorization	
Attacking the logs	Notices you have no logs	
	Puts attacks in the logs to con- fuse logs, log-reading code, or a person reading the logs	

STRIDE

STRIDE classification

THREAT	VIOLATED PROPERTY	THREAT DEFINITION	TYPICAL VICTIMS	EXAMPLES
Information Disclosure	Confidentiality	Providing information to someone not authorized to see it	Processes, data stores, data flows	The most obvious example is allowing access to files, email, or databases, but information disclosure can also involve file- names, packets on a network, or the contents of program memory.

Information Disclosure Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Information disclosure against a process	Extracts secrets from error messages	
	Reads the error messages from username/passwords to entire database tables	
	Extracts machine secrets from error cases	Can make defence against memory corruption such as ASLR far less useful
	Extracts business/personal secrets from error cases	
Information disclosure against data stores	Takes advantage of inappropriate or missing ACLs	
	Takes advantage of bad database permissions	
	Finds files protected by obscurity	
	Finds crypto keys on disk (or in memory)	

Information Disclosure Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Information disclosure against data stores	Sees interesting information in filenames	
	Reads files as they traverse the network	
	Gets data from logs or temp files	
	Gets data from swap or other temp storage	
	Extracts data by obtaining device, changing OS	
Information disclosure against a data flow	Reads data on the network	
	Redirects traffic to enable reading data on the network	
	Learns secrets by analysing traffic	
	Learns who's talking to whom by watching the DNS	
	Learns who's talking to whom by social network info disclosure	

STRIDE

STRIDE classification

THREAT	VIOLATED PROPERTY	THREAT DEFINITION	TYPICAL VICTIMS	EXAMPLES
Denial of Service	Availability	Absorbing resources needed to provide service	Processes, data stores, data flows	A program that can be tricked into using up all its memory, a file that fills up the disk, or so many network connections that real traffic can’t get through

Denial-of-Service Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
DoS against a process	Absorbs memory (RAM or disk)	
	Absorbs CPU	
	Uses process as an amplifier	
DoS against a data store	Fills data store up	
	Makes enough requests to slow down the system	
DoS against a data flow	Consumes network resources	

STRIDE

STRIDE classification

THREAT	VIOLATED PROPERTY	THREAT DEFINITION	TYPICAL VICTIMS	EXAMPLES
Elevation of Privilege	Authorization	Allowing someone to do something they're not authorized to do	Processes	Allowing a normal user to execute code as admin; allowing a remote person without any privileges to run code.

Elevation of Privilege Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Elevation of privilege against a process by corrupting the process	Send inputs that the code doesn’t handle properly	These errors are very com- mon, and are usually high impact
	Gains access to read or write memory inappropriately	Writing memory is (hope- fully obviously) bad, but reading memory can enable further attacks
Elevation through missed authorization checks		
Elevation through buggy authorization checks		Centralizing such checks makes bugs easier to manage
Elevation through data tampering	Modifies bits on disk to do things other than what the authorized user intends	

STRIDE Variants

STRIDE-per-Element (Microsoft)

Observation: certain threats are more prevalent with certain elements of a diagram

- E.g., a data store is unlikely to spoof another data store

		S	T	R	I	D	E
VICTIM	External Entity	X		X			
	Process	X	X	X	X	X	X
	Data Flow		X		X	X	
	Data Store		X	?	X	X	

STRIDE Variants

STRIDE-per-Iteration

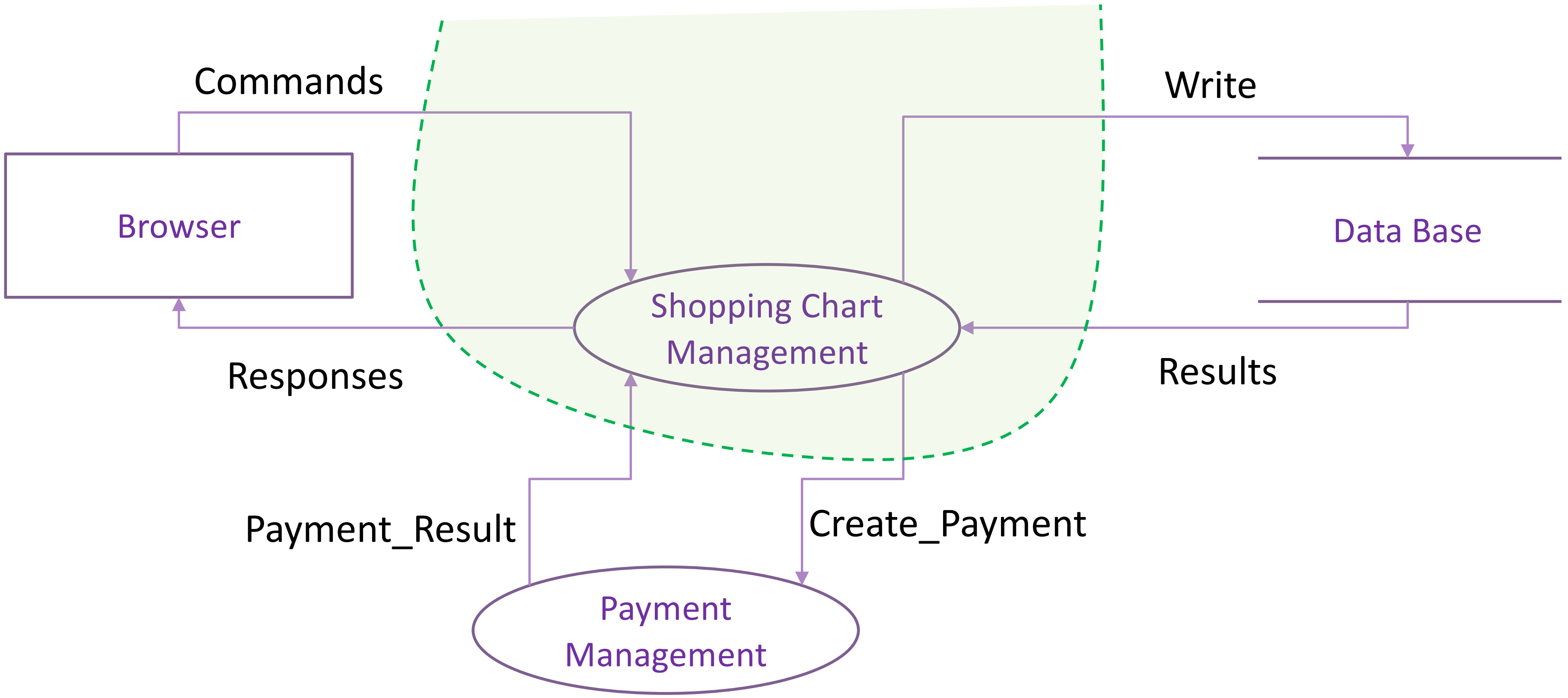
It is an approach to threat enumeration that considers tuples of *(origin, destination, interaction)* and enumerates threats against them

#	ELEMENT	INTERACTION	S	T	R	I	D	E
1	Process 1	Process 1 – Data Store 1	X			X		
2	Process 1	Data Store 1 - Process 1	X	X			X	X
	Data Store 1	…			X			
	Data Flow k							
n	External Entity							

For each Interaction, list all the possible threats in the identified category

Example

Let us consider the following system



Example

#	ELEMENT	INTERACTION	S	T	R	I	D	E
1	Shopping Chart Management	Process has outbound data flow to data store (i.e., write)	X			X		
2		Process sends output to another process	X		X	X	X	X
i	Data Flow (Payment/ Result)	Crosses machine boundary		X		X	X	
j	Data Base	Process has outbound data flow to data store		X	X	X	X	
k	Browser	External interactor passes input to process	X		X	X		

Example

#	ELEMENT	INTERACTION	S	T	R	I	D	E
1	Shopping Chart Management	Process has outbound data flow to data store (i.e., write)	X			X		
2		Process sends output to another process	X		X	X	X	X
i	Data Flow (Payment/ Result)			X		X	X	
j	Data Base	Process has outb e		X	X	X	X	
k	Browser	External interact	X		X	X		

Database is spoofed and Shopping Chart Management writes to the wrong place

Shopping Chart Management writes information in Database which should not be there (e.g., passwords).

Example

#	ELEMENT	INTERACTION	S	T	R	I	D	E
1	Shopping Chart Management	Process has outbound data flow to data store (i.e., write)	X			X		
2		Process sends output to another process	X		X	X	X	X
i	Data Flow (Payment/ Result)	Crosses machine boundary		X		X	X	
j	Data Base	flow to data store		X	X	X	X	
k	Browser	External interactor pa	X		X	X		

Data flow is modified by MITM attack

The contents of the data flow flow are sniffed on the wire

The data flow is interrupted by an external entity (e.g., messing with TCP sequence numbers)

Example

#	ELEMENT	INTERACTION	S	T	R	I	D	E
1	Shopping Chart Management	Process has store (i.e., write)	X			X		
2		ds output to another process	X		X	X	X	X
i	Data	Crosses machine boundary		X		X	X	
j	Data Base	Process has outbound data flow to data store		X	X	X	X	
k	Browser	nput to process	X		X	X		

Database reveals information

Shopping Chart Management claims not to have written to database

Database is corrupted

Database cannot be written to

Example

#	ELEMENT	INTERACTION	S	T	R	I	D	E
1	Shopping Chart Management	Process has outbound da (i.e., write)				X		
2		Process sends output to			X	X	X	X
i	Data Flo (Paymen	e boundary		X		X	X	
j		has outbound data flow to data store		X	X	X	X	
k	Browser	External interactor passes input to process	X		X	X		

Shopping Chart Management not authorized to receive data

Shopping Chart Management claims not to have received the data

Shopping Chart Management is confused about the identity of the browser

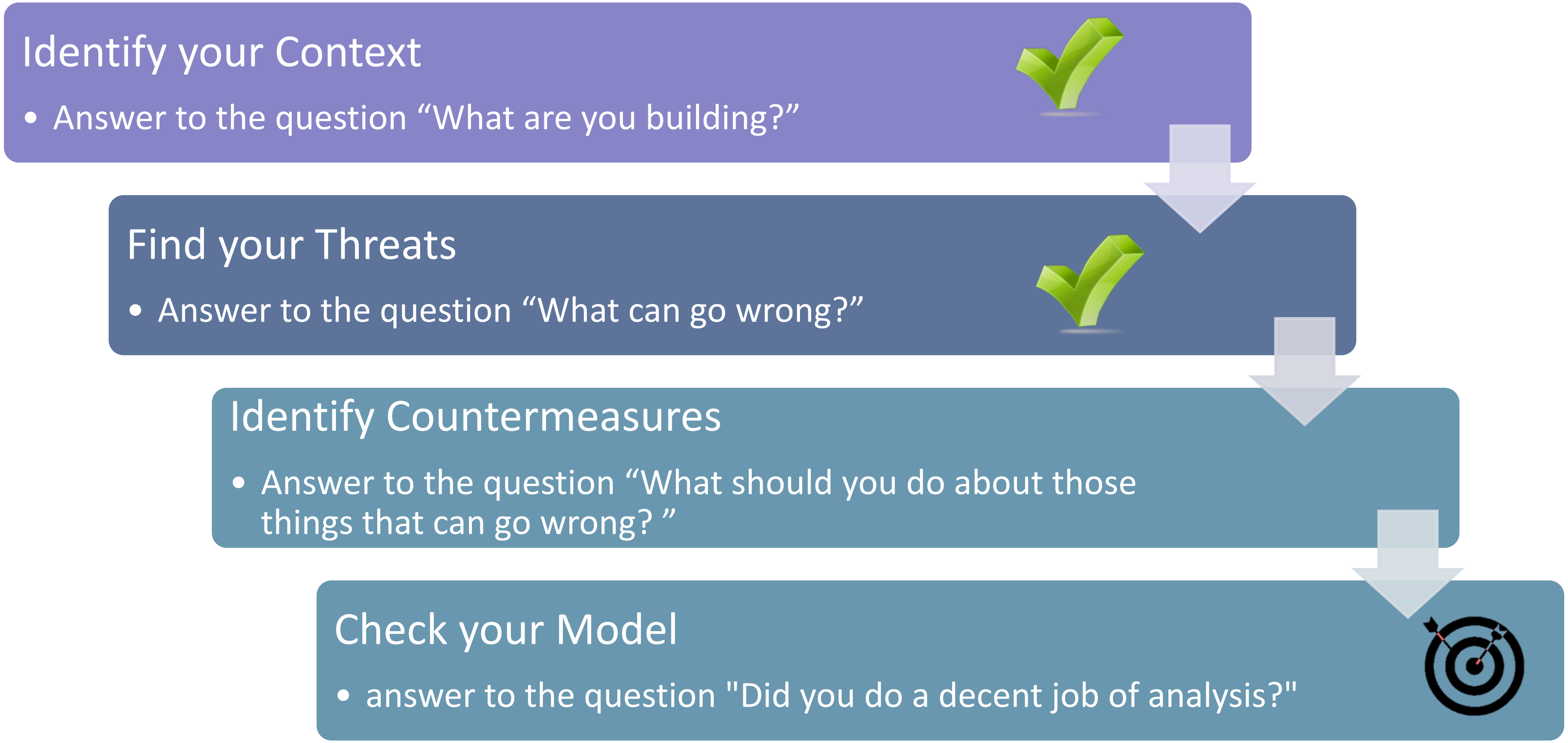
STRIDE Variants

DESIST

DESIST stands for

- Dispute (replace Repudiation)
- Elevation of privilege
- Spoofing
- Information disclosure
- Service denial (replace Denial of Service)
- Tampering

Recap
Basic Steps for Threat Modelling



Identify your Context ✓

- Answer to the question “What are you building?”

Find your Threats ✓

- Answer to the question “What can go wrong?”

Identify Countermeasures

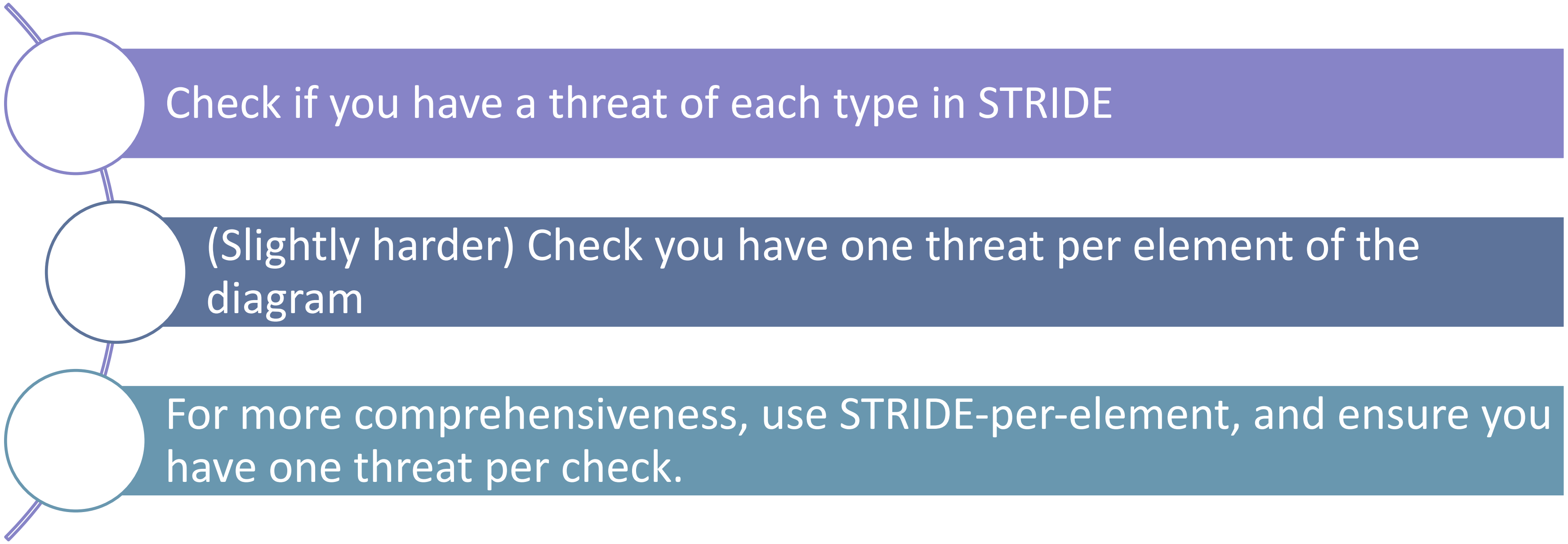
- Answer to the question “What should you do about those things that can go wrong? ”

Check your Model

- answer to the question "Did you do a decent job of analysis?"

Check your STRIDE-driven model

There are three ways to judge whether you have done finding threats with STRIDE



Check if you have a threat of each type in STRIDE

(Slightly harder) Check you have one threat per element of the diagram

For more comprehensiveness, use STRIDE-per-element, and ensure you have one threat per check.

Not having met these criteria will tell you that you have not done, but having met them is not a guarantee of completeness

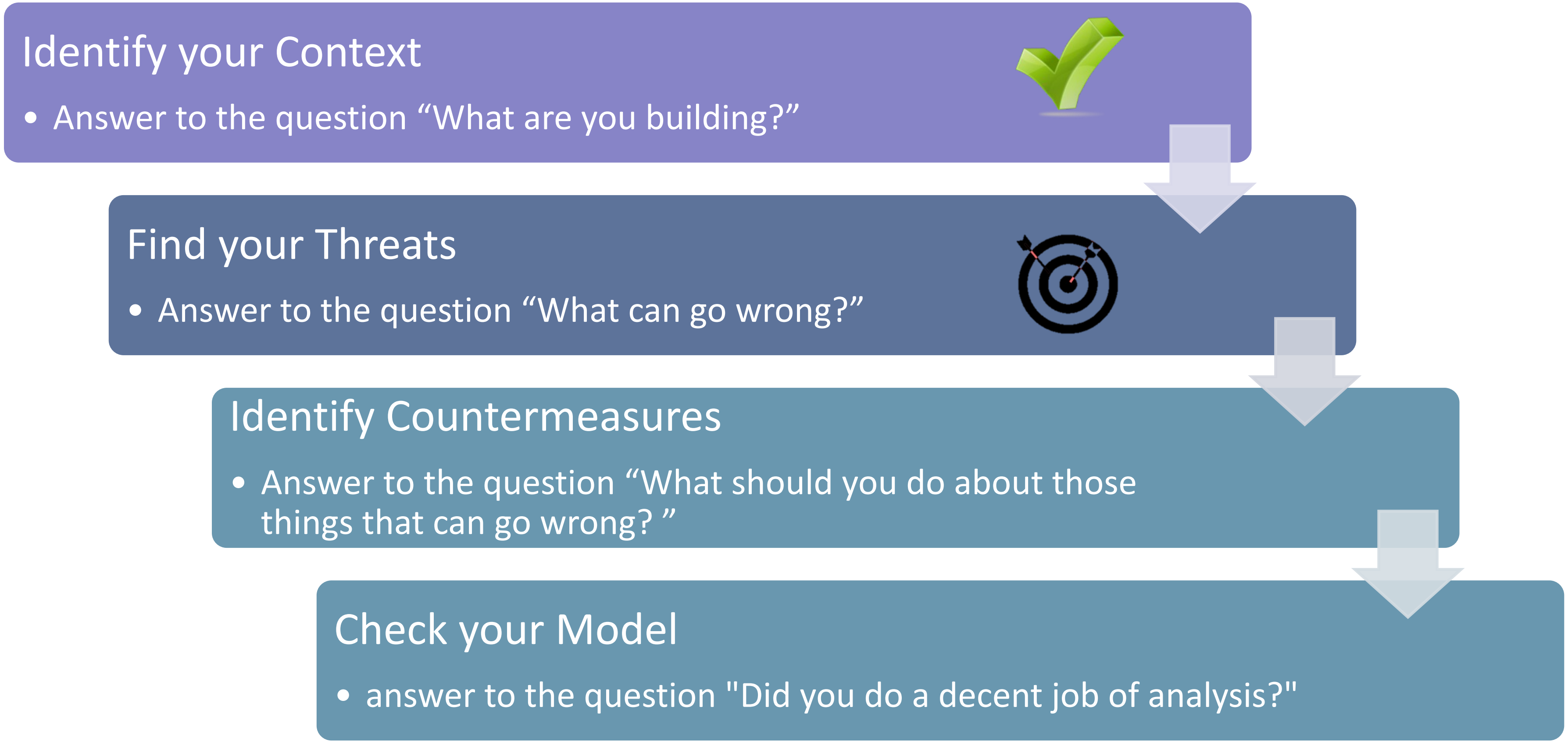
Observations

1. When using STRIDE you are just enumerating the things that might go wrong
   - The exact mechanisms for how it can go wrong are something you can analyse later

2. It can be useful to record all possible attacks, even if there is a mitigation in place,
   - E.g., you may list as that “Someone could modify the management tables” and someone may complain “No, they can’t because...”
   - that mitigation is a testable feature, and you should ensure that you have a test case for it

3. STRIDE is not a taxonomy or a classification mechanism
   - It is easy to find things that are hard to match with just one STRIDE criteria

Recap
Basic Steps for Threat Modelling



Identify your Context

- Answer to the question “What are you building?”

Find your Threats

- Answer to the question “What can go wrong?”

Identify Countermeasures

- Answer to the question “What should you do about those things that can go wrong? ”

Check your Model

- answer to the question "Did you do a decent job of analysis?"

Attack trees

What is an attack tree?

- A way of thinking and describing security of systems and subsystems
- A way of building an automatic database that describes the security of a system
- A way of capturing expertise, and reusing it
- A way of making decisions about how to improve security, or the effects of a new attack on security

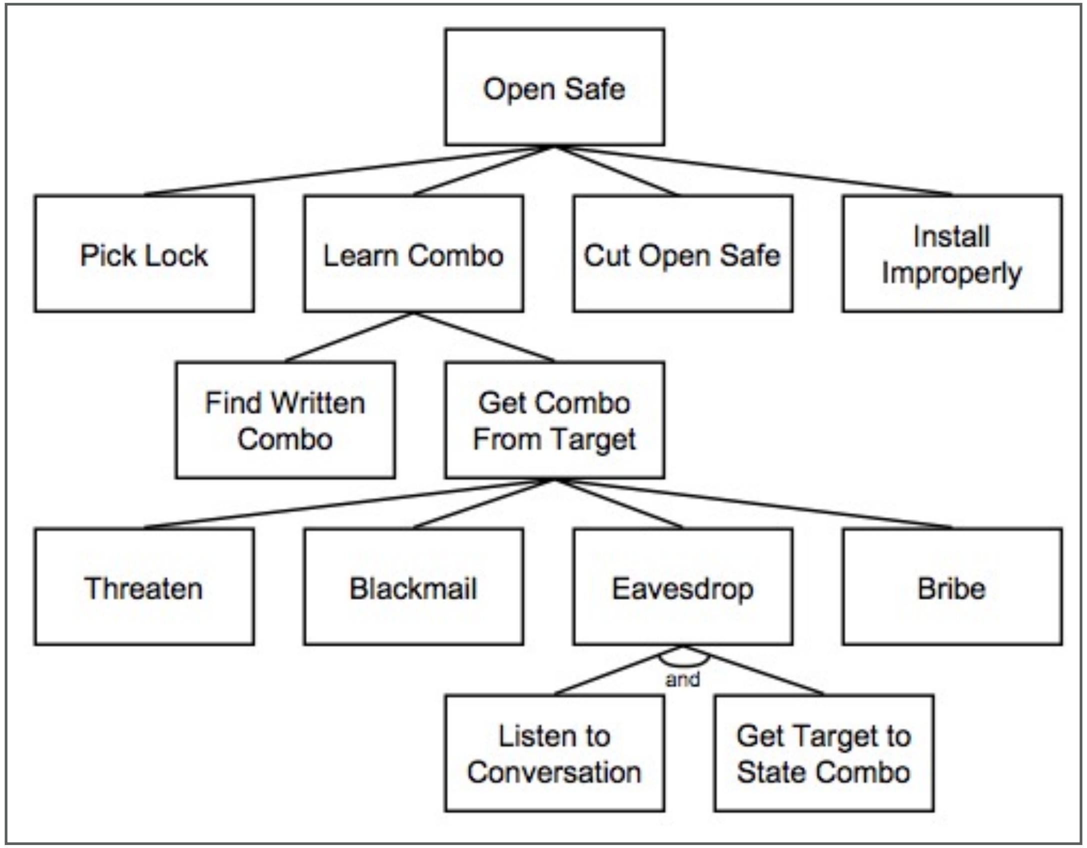
Part of these slides from: B. Schneier "Attack Trees"

Attack trees

What is an attack tree?

- Represents attacks and countermeasures as a tree structure
- Root node is the goal of the attack
    - In a complex system there are probably several trees, each representing a different goal
- Leaf nodes represent specific attacks used to reach the goal

Attack tree Example



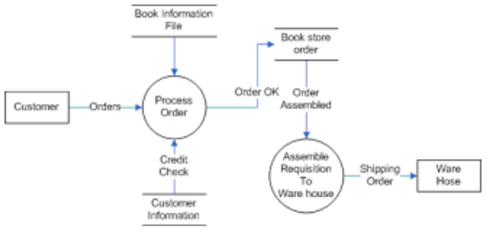
Attack trees

There are three ways you can use attack trees to enumerate threats

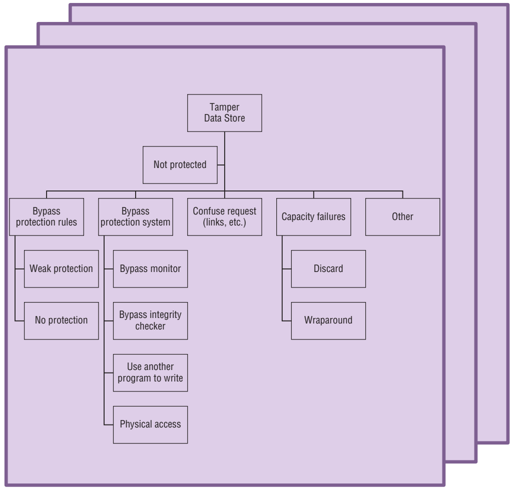
1. use an attack tree someone else created to help you find threats
2. create a tree to help you think through threats for a project you’re working on
3. create trees with the intent that others will use them

⚠ Creating new trees for general use is challenging, even for security experts

Using Attack Trees to Find Threats

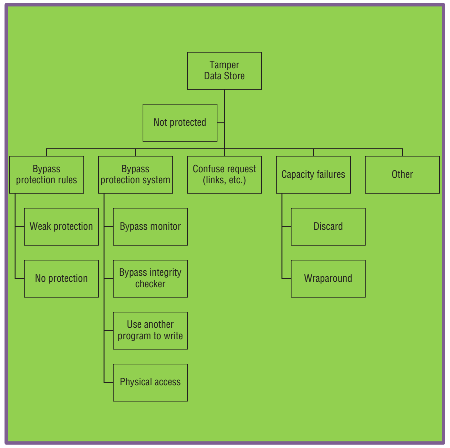


System DFD



Existing ATs

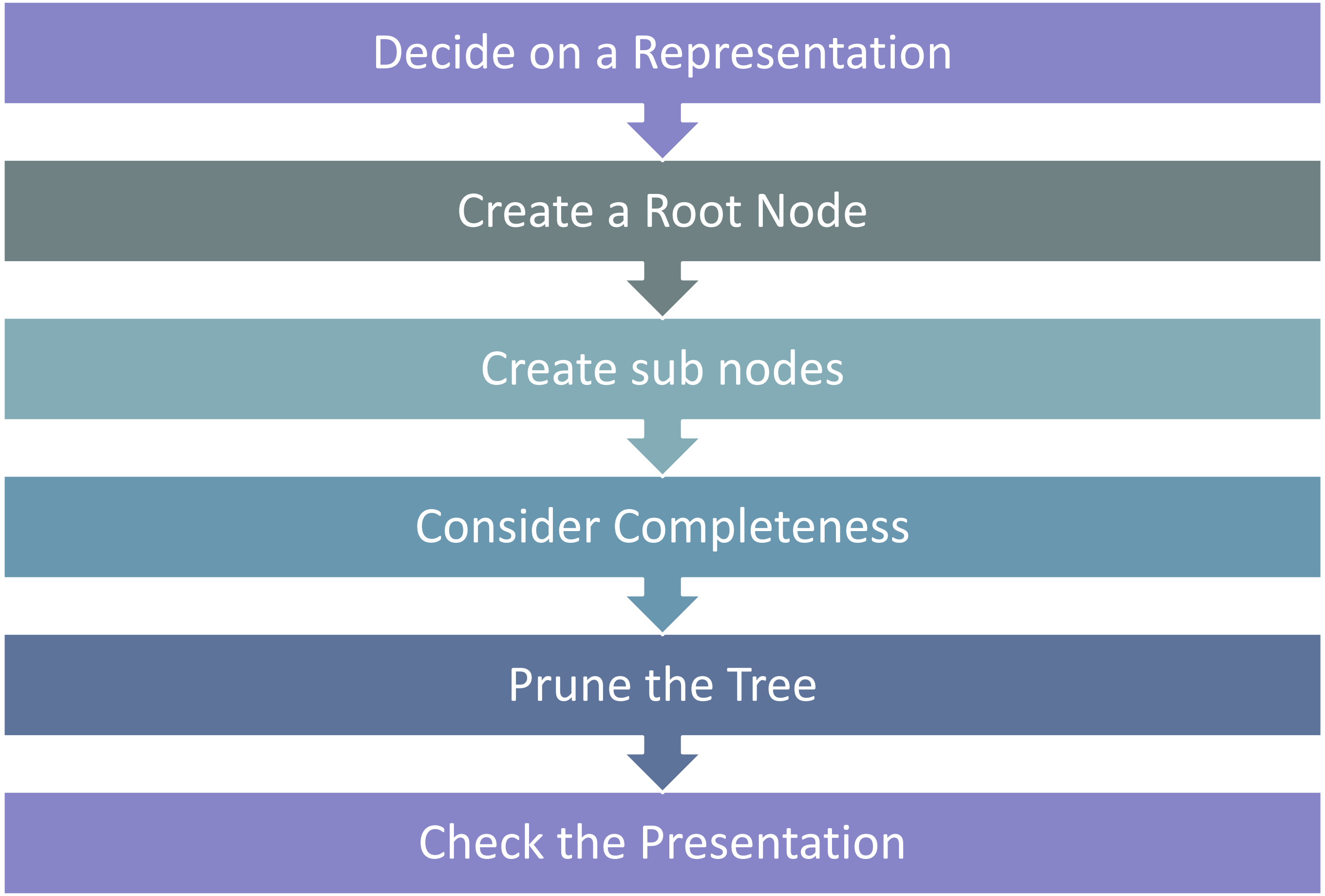
Identify Specific Threats



The elicitation task is done trough iteration over each node in the tree considering if that issue (or a variant of that issue) impacts your system.

Creating New Attack Trees

A project-specific tree is a way to organize your thinking about threats



Decide on a Representation

Create a Root Node

Create sub nodes

Consider Completeness

Prune the Tree

Check the Presentation

Attack trees Representations

AND trees where the state of a node depends on all of the nodes below it being true

- represent different steps in achieving a goal
- E.g., to enter through a window you need to break the window AND climb through the opening

OR trees where a node is true if any of its sub nodes are true

- represent different ways to achieve the same goal
- E.g., to break into a house you can either pick the lock OR break a window

Create a Root Node

The root node can be

- the component that prompts the analysis
  - the sub nodes should be labelled with what can go wrong for the node
- an adversary’s goal
  - the sub nodes should be labelled with ways to achieve that goal
- a problematic state

SUGGESTIONS

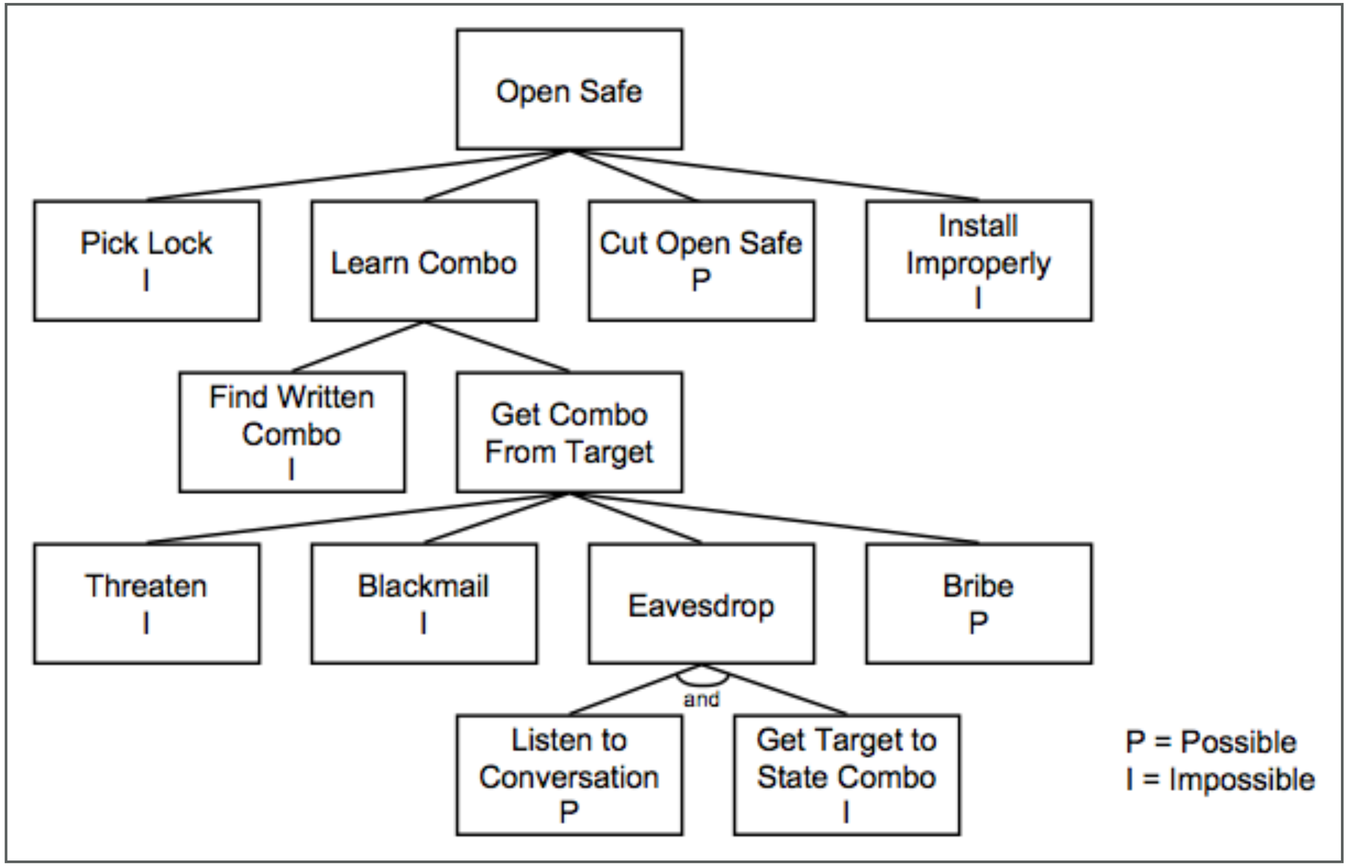
- Create a root node with an attacker goal or high-impact action.
- Use OR trees.
- Draw them into a grid that the eye can track linearly

Attack trees

You can assign values to leaf nodes

- Values may be used to characterize the potential attack in different ways
- The simples values are boolean
  - Possible vs. Impossible

Attack trees



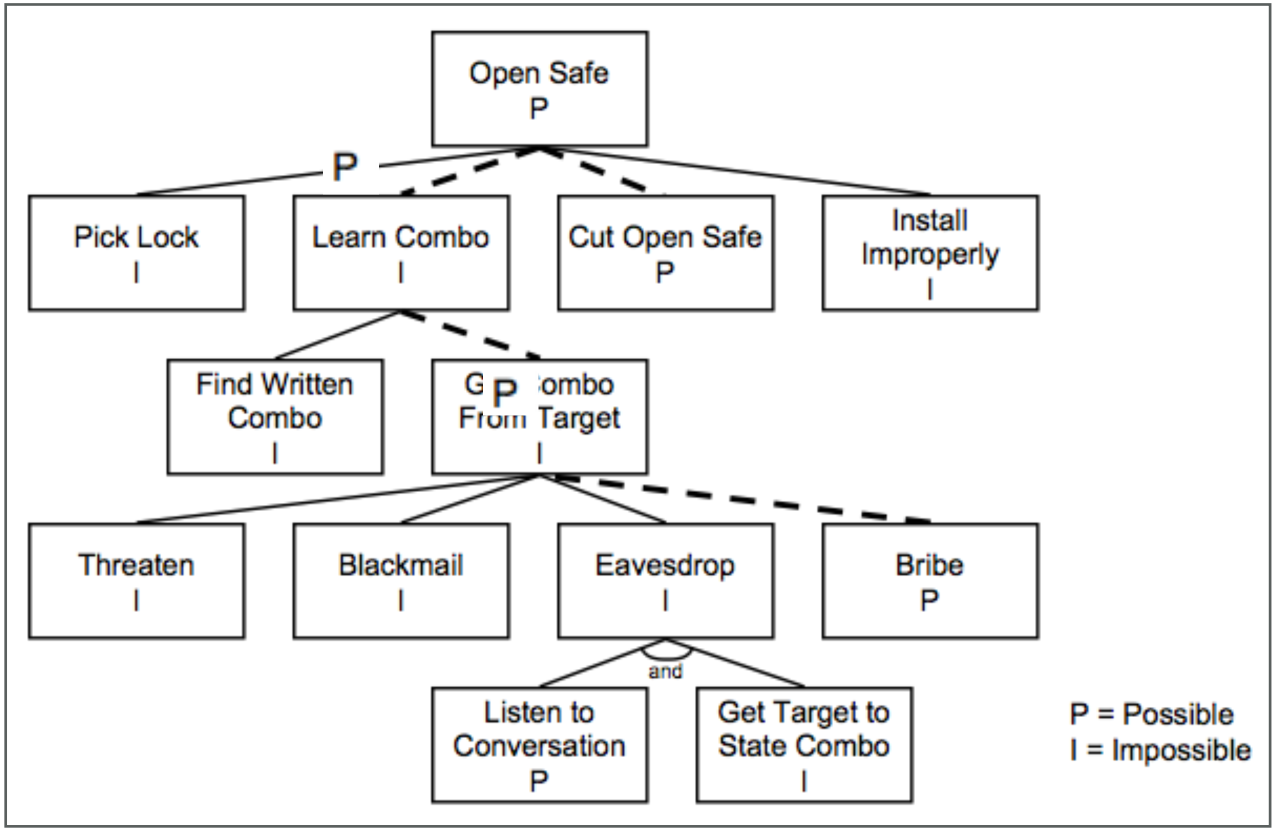
Attack trees

A node’s value is a function of its children’s

AND/OR nodes require different calculations

Start from each root node and calculate while moving toward the root

Attack trees

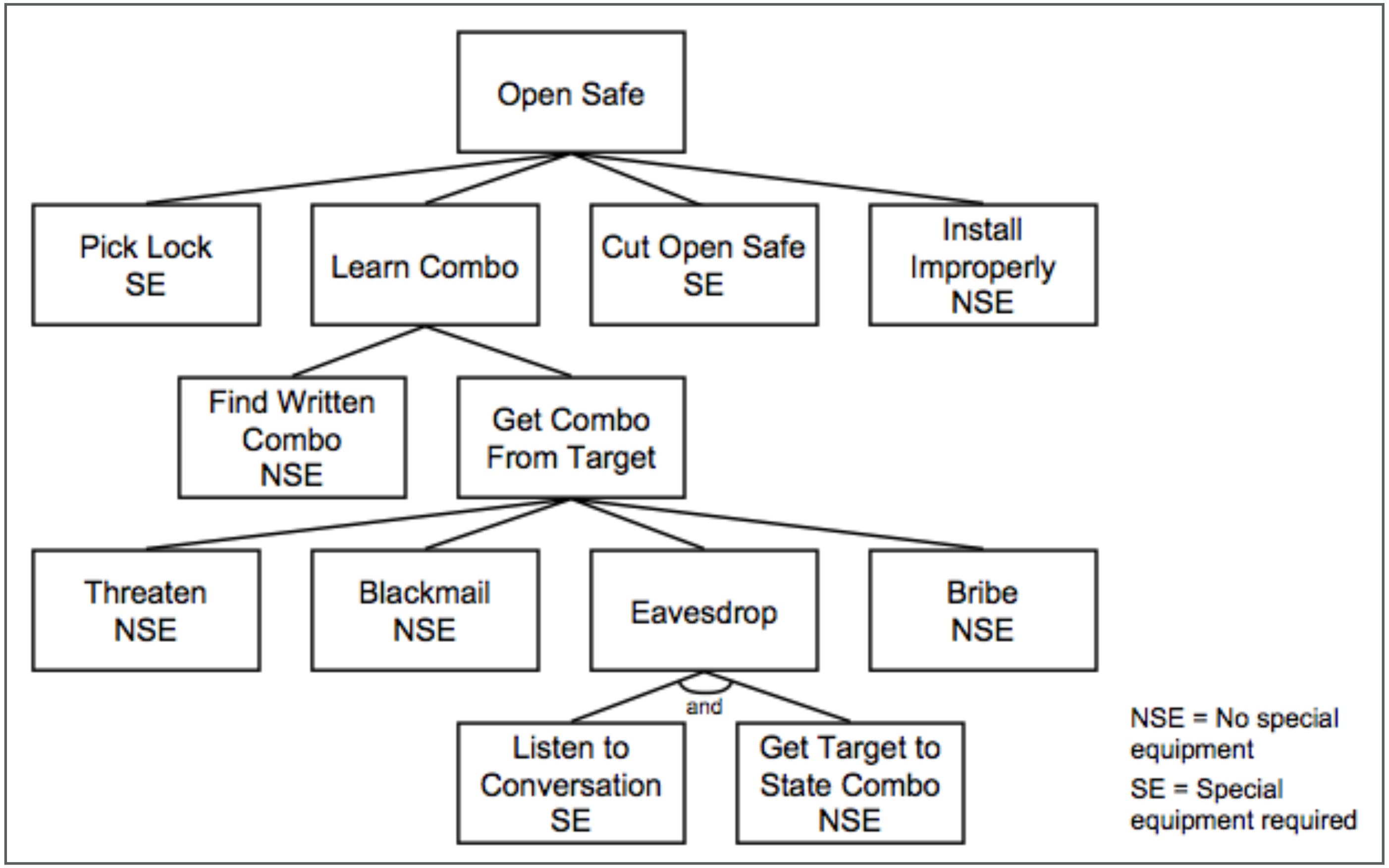


Attack trees

Any boolean value can be codified in the leaf nodes and then used to prune the tree

- Easy/Complex
- Expensive/Cheap
- Intrusive/Non-intrusive
- Legal/Illegal
- Special Equipment required or not required

Attack trees



Open Safe

Pick Lock SE

Learn Combo

Cut Open Safe SE

Install Improperly NSE

Find Written Combo NSE

Get Combo From Target

Threaten NSE

Blackmail NSE

Eavesdrop

Bribe NSE

and

Listen to Conversation SE

Get Target to State Combo NSE

NSE = No special equipment

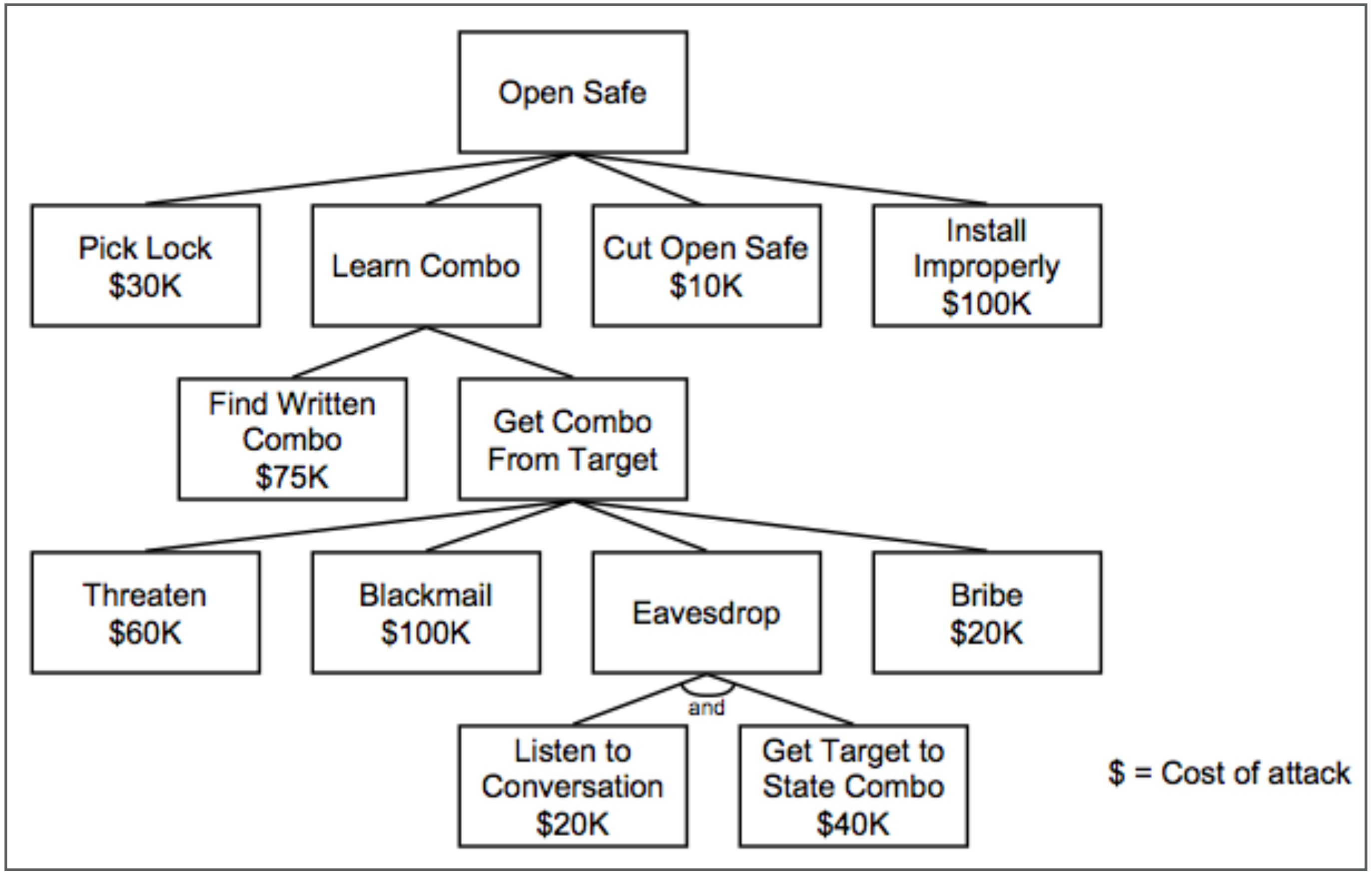
SE = Special equipment required

Attack trees

You can also label leaf nodes with continuous values

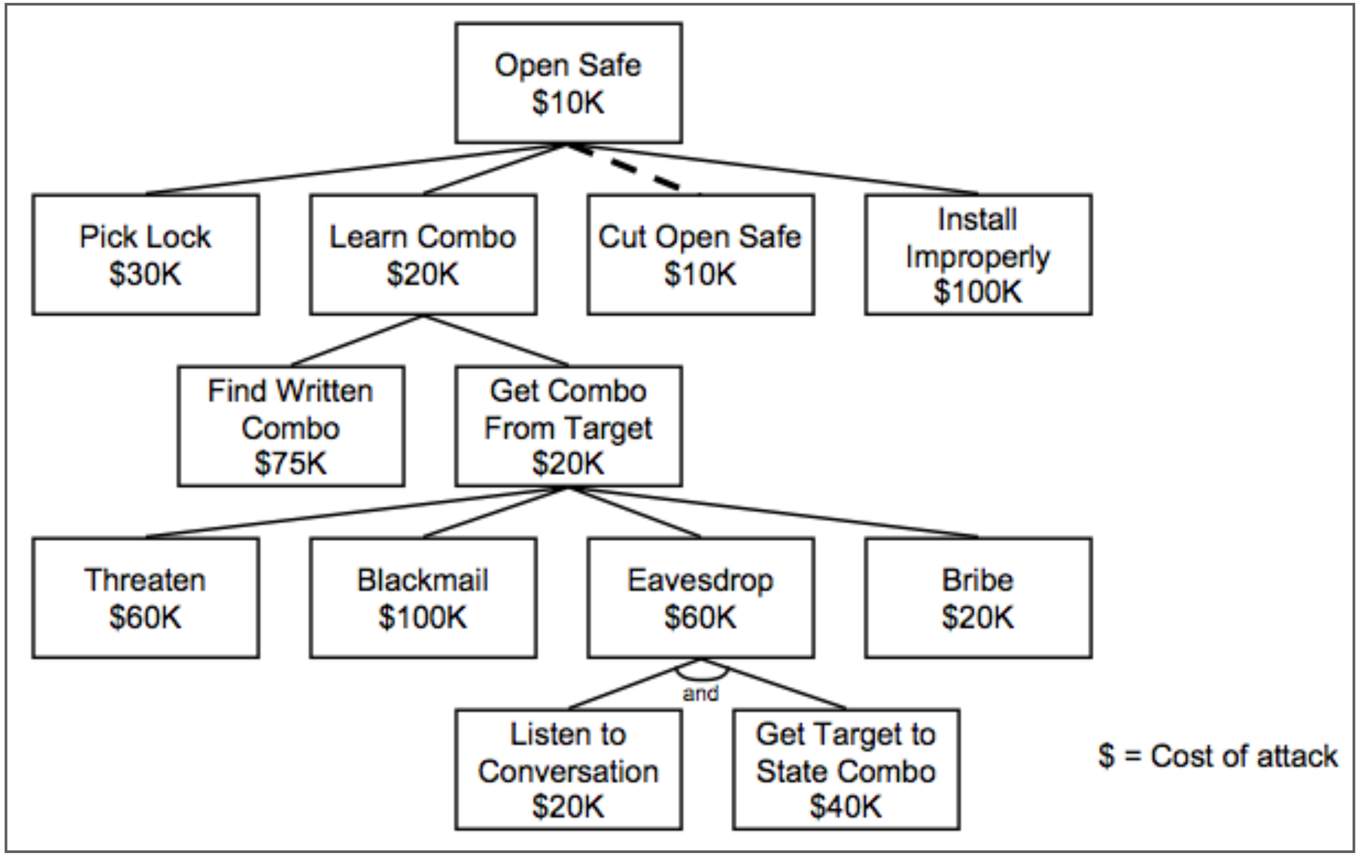
- Cost of the attack
- Cost of defense
- Time to achieve
- Resources needed to attack
- Probability of attack success
- Likelihood that an attacker will try a given attack

Attack trees



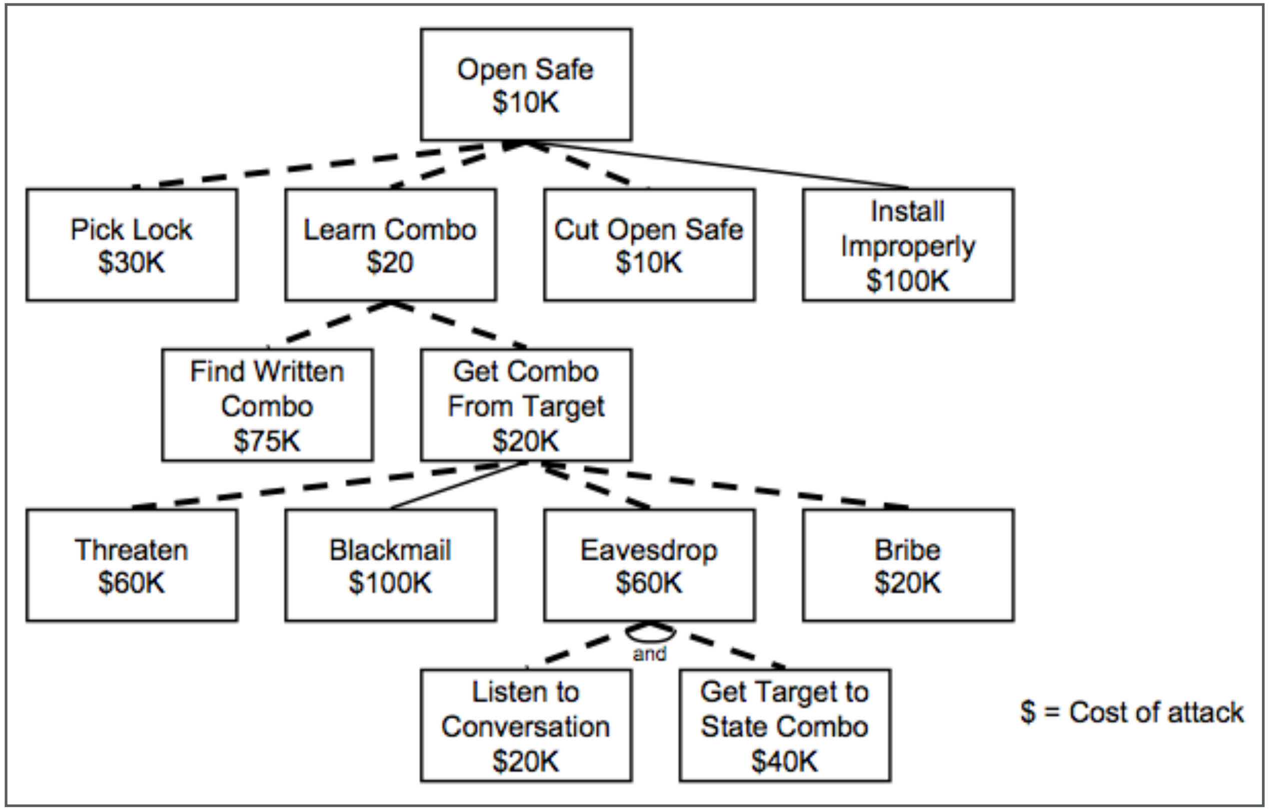
Attack trees

Using these values you could for example calculate the cheapest attack



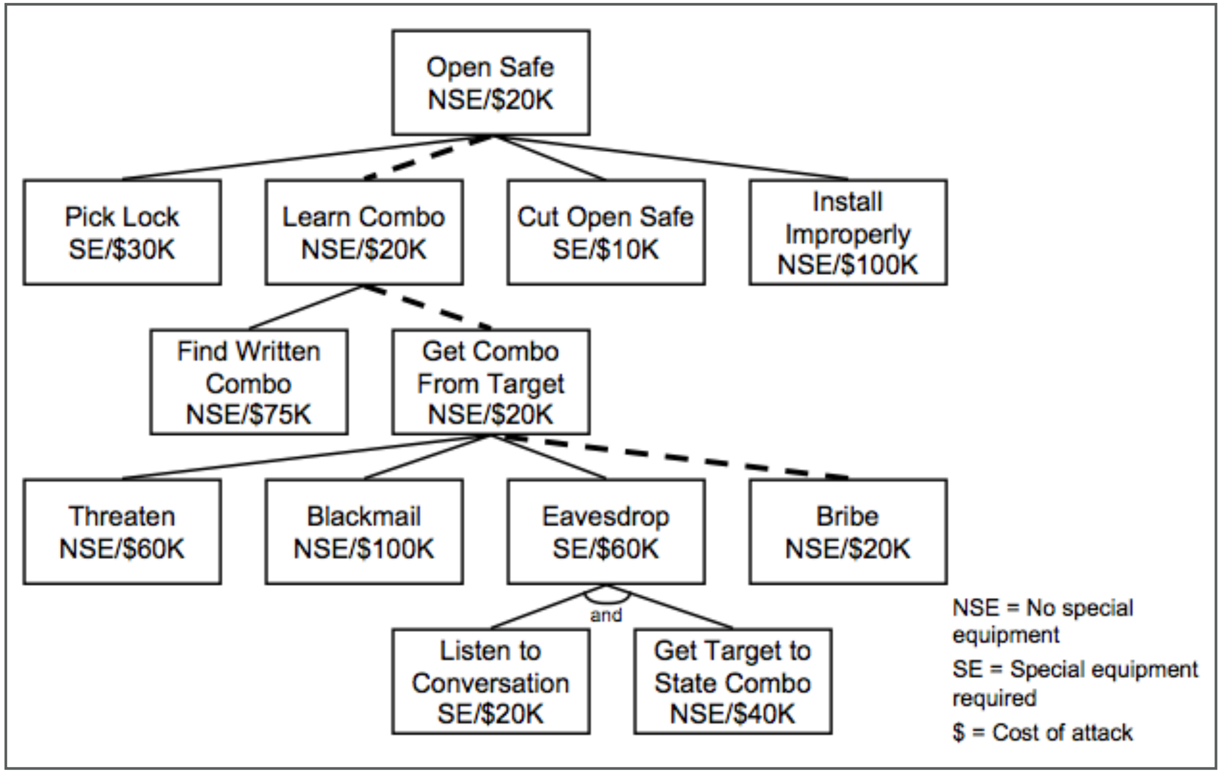
Attack trees

Which attacks will cost less than 100$?



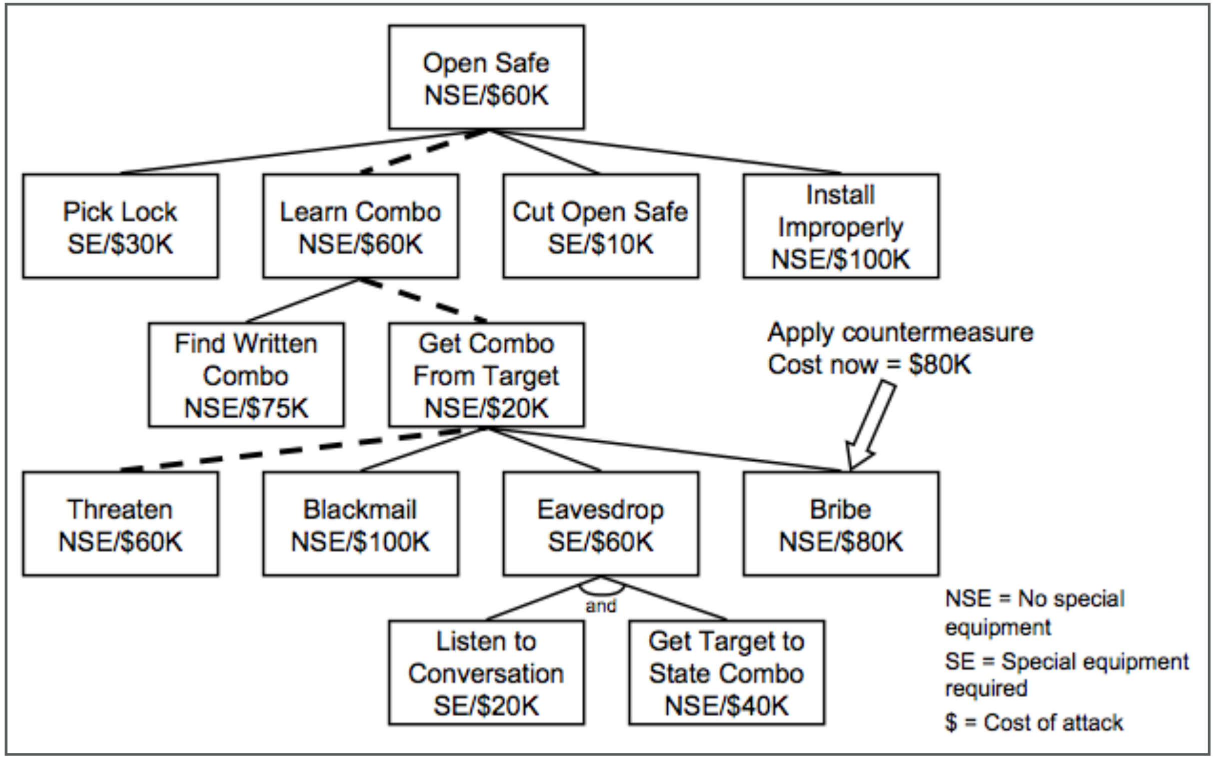
Attack trees

You can also combine info: cheapest without special equipment?



Attack trees

Contextualize countermeasures in the tree



Apply countermeasure
Cost now = $80K

NSE = No special equipment

SE = Special equipment required

$ = Cost of attack

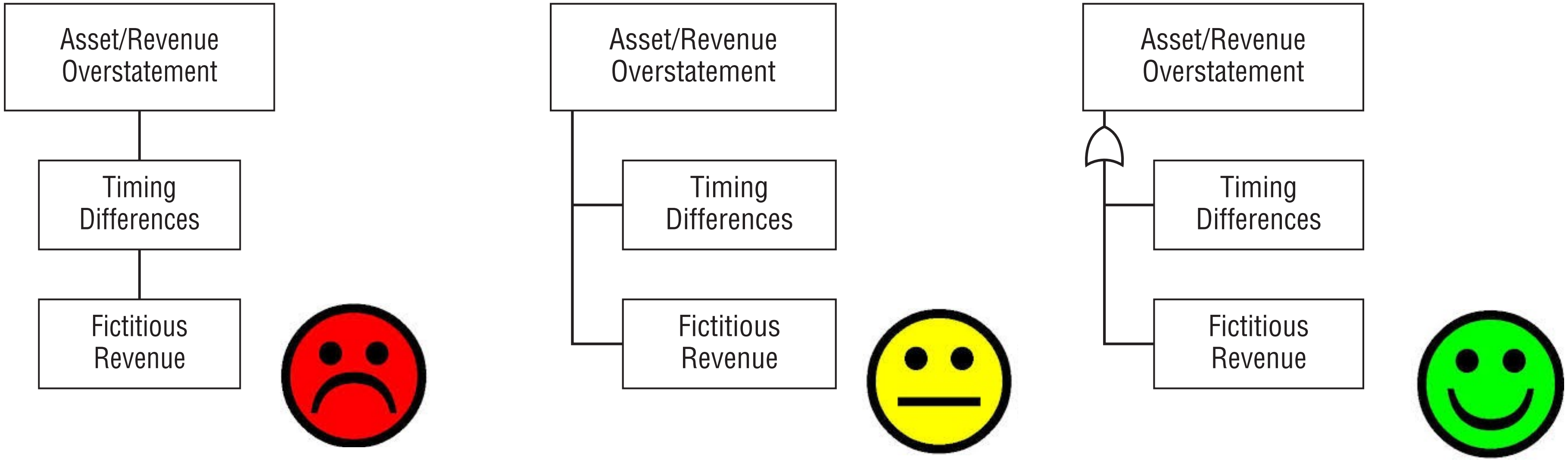
Human-Viewable Representations

Attack trees can be drawn graphically or shown in outline form

In any case, they should be compact (no more than 1 page)

- If your tree is too large, split it in to multiple trees

Graphical representations must be information-rich and communicative



Source: A. Shostack "Threat modeling. Design for security", Wiley, 2014

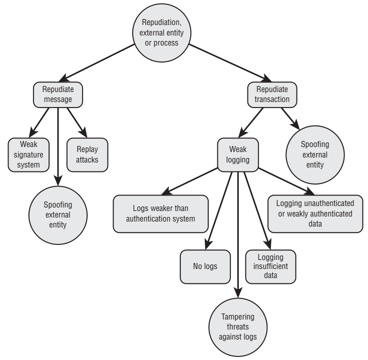
Human-Viewable Representations

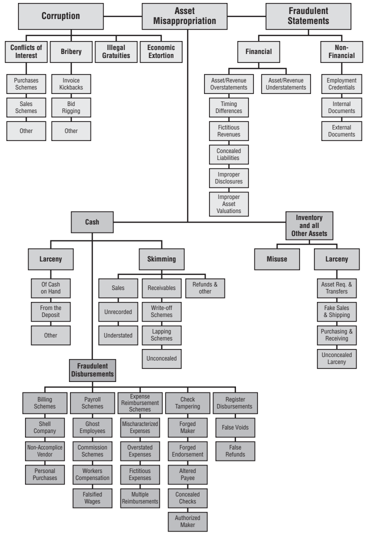
Attack trees can be drawn graphically or shown in outline form

In any case, they should be compact (no more than 1 page)

- If your tree is too large, split it in to multiple trees

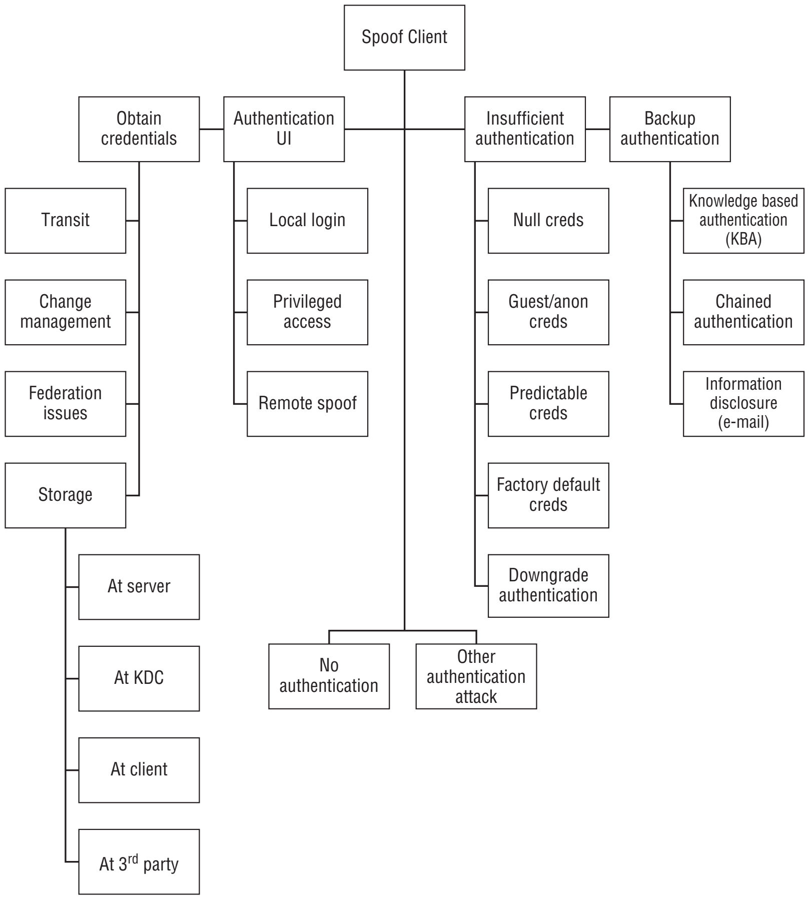
Graphical representations must be information-rich and communicative


Repudiation, external entity or process
Repudiate message
Repudiate transaction
Weak signature system
Replay attacks
Spoofing external entity
Weak logging
Spoofing external entity
Logs weaker than authentication system
Logging unauthenticated or weakly authenticated data
No logs
Logging insufficient data
Tampering threats against logs



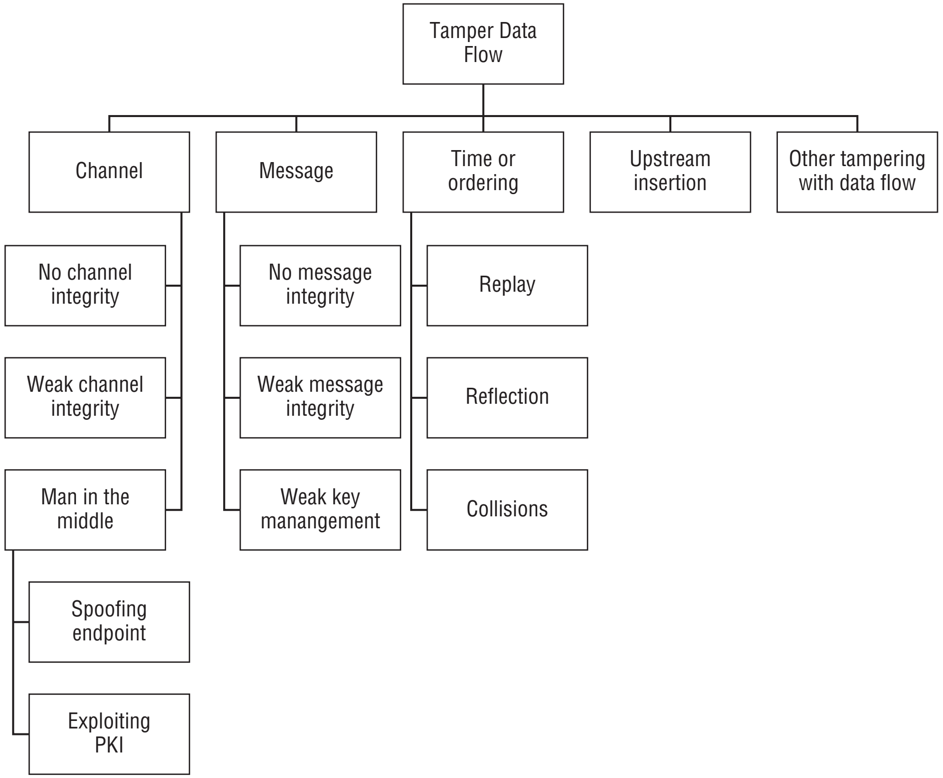
Source: A. Shostack “Threat modeling. Design for security”, Wiley, 2014

Attack trees - EXAMPLES



Source: A. Shostack "Threat modeling. Design for security", Wiley, 2014

Attack trees - EXAMPLES



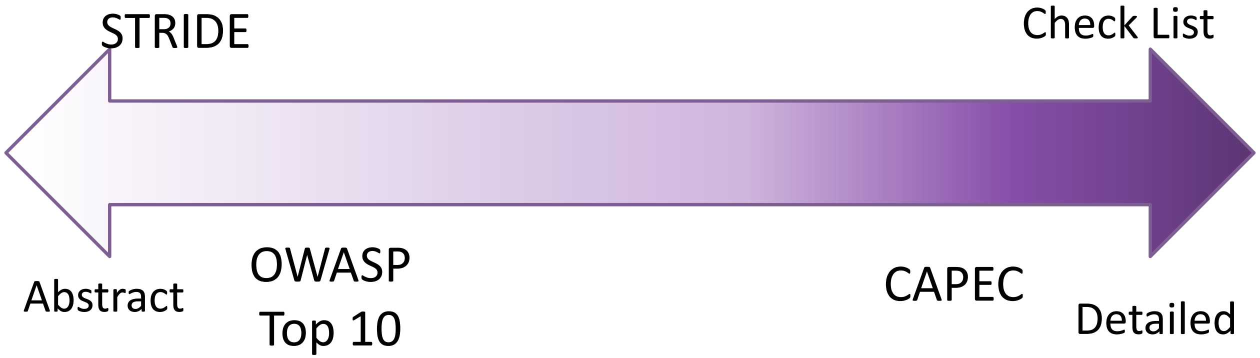
Source: A. Shostack "Threat modeling. Design for security", Wiley, 2014

Attack Libraries

A library of attacks can be a useful tool for finding threats against the system you're building.

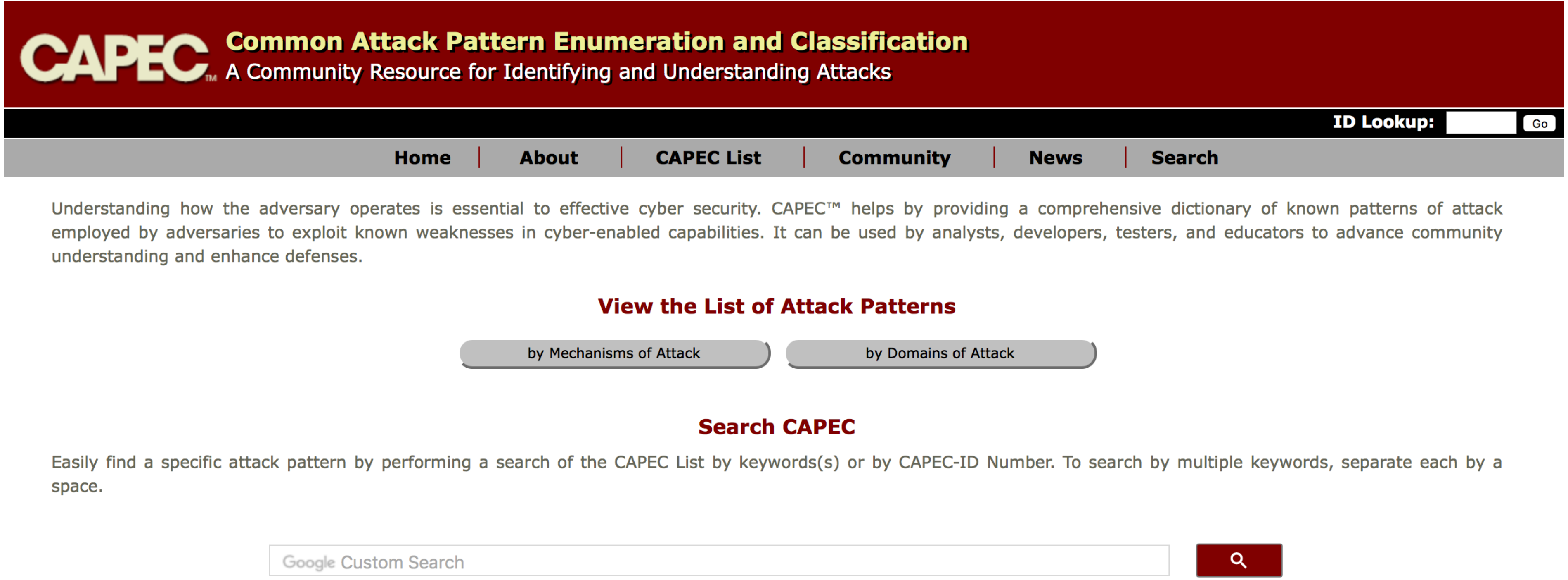
Different libraries address different goals depending on their focus in terms of

- Audience
- Scope
- Detail vs abstraction



RECAP: MITRE CAPEC[1]

***CAPEC** is a publicly available catalogue of attack patterns along with a comprehensive schema and classification taxonomy created to assist in the building of secure software*



CAPEC™
Common Attack Pattern Enumeration and Classification
A Community Resource for Identifying and Understanding Attacks
ID Lookup:
Go
Home | About | CAPEC List | Community | News | Search

Understanding how the adversary operates is essential to effective cyber security. CAPEC™ helps by providing a comprehensive dictionary of known patterns of attack employed by adversaries to exploit known weaknesses in cyber-enabled capabilities. It can be used by analysts, developers, testers, and educators to advance community understanding and enhance defenses.

View the List of Attack Patterns

Search CAPEC

Easily find a specific attack pattern by performing a search of the CAPEC List by keywords(s) or by CAPEC-ID Number. To search by multiple keywords, separate each by a space.

[1] https://capec.mitre.org

RECAP: CAPEC Entry Details

Typical severity

A description, including:
- Summary
- Attack execution flow

Prerequisites

Method(s) of attack

Examples

Attacker skills or knowledge required

Resources required

Probing techniques

Indicators/warnings of attack

Solutions and mitigations

Attack motivation/consequences

Vector

Payload

Relevant security requirements, principles and guidance

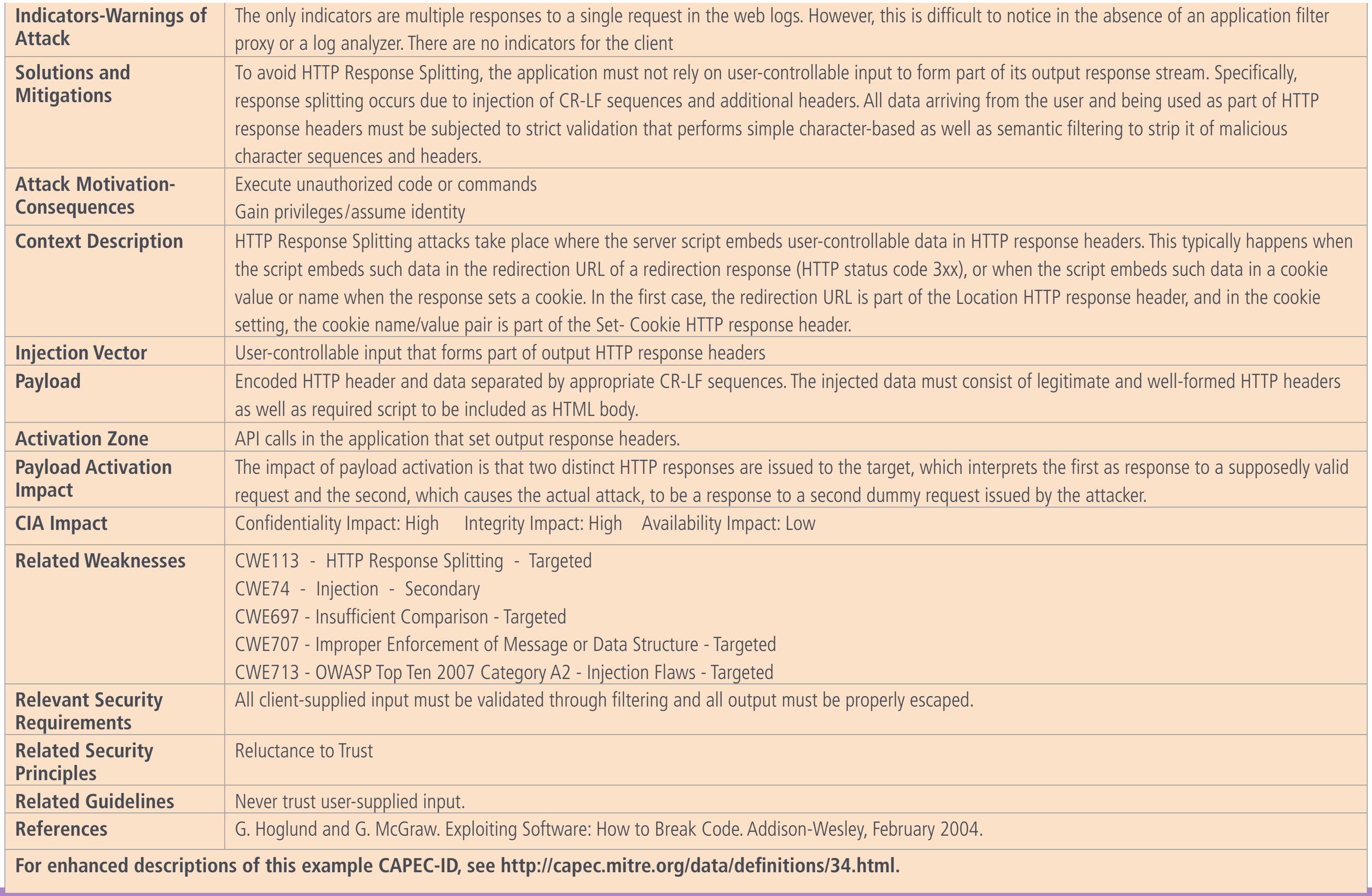
Technical context

A variety of bookkeeping fields (identifier, related attack patterns and vulnerabilities, change history, etc.)

CAPEC Attack Pattern Example

Name	HTTP Response Splitting
Typical Severity	High
Description	HTTP Response Splitting causes a vulnerable web server to respond to a maliciously crafted request by sending an HTTP response stream such that it gets interpreted as two separate responses instead of a single one. This is possible when user-controlled input is used unvalidated as part of the response headers. An attacker can have the victim interpret the injected header as being a response to a second dummy request, thereby causing the crafted contents to be displayed and possibly cached. To achieve HTTP Response Splitting on a vulnerable web server, the attacker: 1. Identifies the user-controllable input that causes arbitrary HTTP header injection. 2. Crafts a malicious input consisting of data to terminate the original response and start a second response with headers controlled by the attacker. 3. Causes the victim to send two requests to the server. The first request consists of maliciously crafted input to be used as part of HTTP response headers and the second is a dummy request so that the victim interprets the split response as belonging to the second request.
Attack Prerequisites	User-controlled input used as part of HTTP header Ability of attacker to inject custom strings in HTTP header Insufficient input validation in application to check for input sanity before using it as part of response header
Typical Likelihood of Exploit	Medium
Methods of Attack	Injection Protocol Manipulation
Examples-Instances	In the PHP 5 session extension mechanism, a user-supplied session ID is sent back to the user within the Set-Cookie HTTP header. Since the contents of the user-supplied session ID are not validated, it is possible to inject arbitrary HTTP headers into the response body. This immediately enables HTTP Response Splitting by simply terminating the HTTP response header from within the session ID used in the Set-Cookie directive. CVE-2006-0207
Attacker Skill or Knowledge Required	High - The attacker needs to have a solid understanding of the HTTP protocol and HTTP headers and must be able to craft and inject requests to elicit the split responses.
Resources Required	None
Probing Techniques	With available source code, the attacker can see whether user input is validated or not before being used as part of output. This can also be achieved with static code analysis tools If source code is not available, the attacker can try injecting a CR-LF sequence (usually encoded as %0d%0a in the input) and use a proxy such as Paros to observe the response. If the resulting injection causes an invalid request, the web server may also indicate the protocol error.

CAPEC Attack Pattern Example



Indicators-Warnings of Attack	The only indicators are multiple responses to a single request in the web logs. However, this is difficult to notice in the absence of an application filter proxy or a log analyzer. There are no indicators for the client
Solutions and Mitigations	To avoid HTTP Response Splitting, the application must not rely on user-controllable input to form part of its output response stream. Specifically, response splitting occurs due to injection of CR-LF sequences and additional headers. All data arriving from the user and being used as part of HTTP response headers must be subjected to strict validation that performs simple character-based as well as semantic filtering to strip it of malicious character sequences and headers.
Attack Motivation-Consequences	Execute unauthorized code or commands Gain privileges/assume identity
Context Description	HTTP Response Splitting attacks take place where the server script embeds user-controllable data in HTTP response headers. This typically happens when the script embeds such data in the redirection URL of a redirection response (HTTP status code 3xx), or when the script embeds such data in a cookie value or name when the response sets a cookie. In the first case, the redirection URL is part of the Location HTTP response header, and in the cookie setting, the cookie name/value pair is part of the Set- Cookie HTTP response header.
Injection Vector	User-controllable input that forms part of output HTTP response headers
Payload	Encoded HTTP header and data separated by appropriate CR-LF sequences. The injected data must consist of legitimate and well-formed HTTP headers as well as required script to be included as HTML body.
Activation Zone	API calls in the application that set output response headers.
Payload Activation Impact	The impact of payload activation is that two distinct HTTP responses are issued to the target, which interprets the first as response to a supposedly valid request and the second, which causes the actual attack, to be a response to a second dummy request issued by the attacker.
CIA Impact	Confidentiality Impact: High Integrity Impact: High Availability Impact: Low
Related Weaknesses	CWE113 - HTTP Response Splitting - Targeted CWE74 - Injection - Secondary CWE697 - Insufficient Comparison - Targeted CWE707 - Improper Enforcement of Message or Data Structure - Targeted CWE713 - OWASP Top Ten 2007 Category A2 - Injection Flaws - Targeted
Relevant Security Requirements	All client-supplied input must be validated through filtering and all output must be properly escaped.
Related Security Principles	Reluctance to Trust
Related Guidelines	Never trust user-supplied input.
References	G. Hoglund and G. McGraw. Exploiting Software: How to Break Code. Addison-Wesley, February 2004.
For enhanced descriptions of this example CAPEC-ID, see http://capec.mitre.org/data/definitions/34.html.	

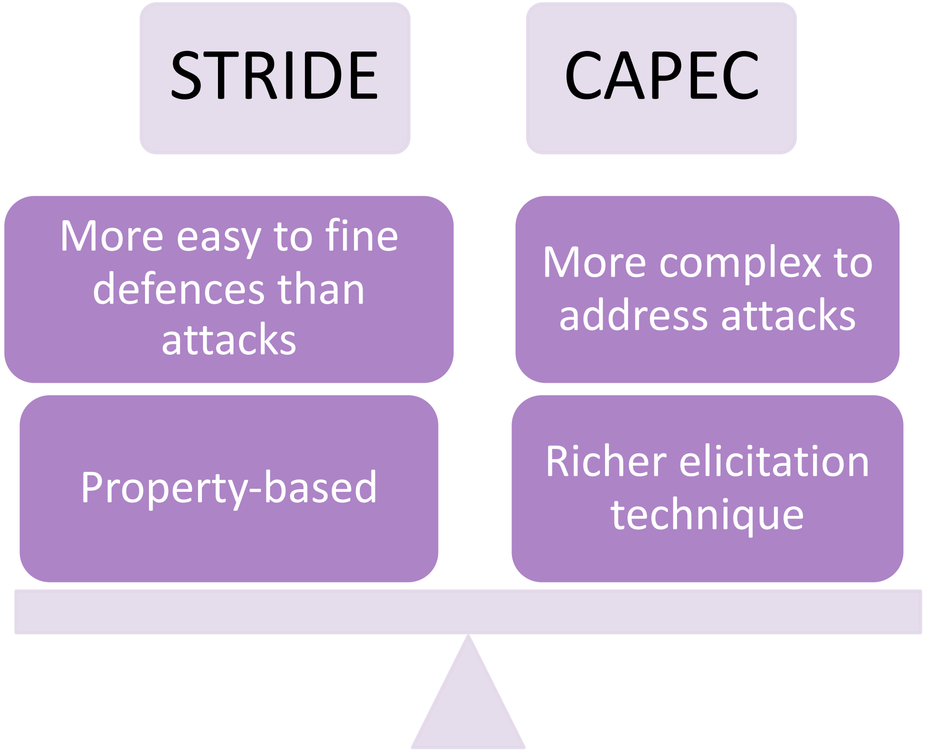
CAPEC

You can use this very structured set of information for threat modelling in a few ways

- review a system being built against either each CAPEC entry or the 15 CAPEC categories.
- reviewing against the individual entries is a large task
  - Using, on average, five minutes for each of the 475 entries, that’s a full 40 hours of work.
- train people about the breadth of threats.
  - create a training class, probably taking a day or more.

CAPEC vs STRIDE

CAPEC is a classification of common attacks, whereas STRIDE is a set of security properties



STRIDE

CAPEC

More easy to fine defences than attacks

More complex to address attacks

Property-based

Richer elicitation technique

OWASP Top 10

The OWASP Top 10 is a powerful awareness document for web application security.

It represents a broad consensus about the most critical security risks to web applications

Project members include a variety of security experts from around the world who have shared their expertise to produce this list

Although the original goal of the OWASP Top 10 project was simply to raise awareness amongst developers and managers, it has become the de facto application security standard

OWASP Top 10 2021

OWASP Top 10 - 2021	
1	Broken Access Control
2	Cryptographic Failure
3	Injection
4	Insecure Design
5	Security Misconfiguration
6	Vulnerable and Outdated Components
7	Identification and Authentication Failures
8	Software and Data Integrity Failures
9	Security Logging and Monitoring Failures
10	Server-Side Request Forgery

OWASP Top 10 2021 vs 2017

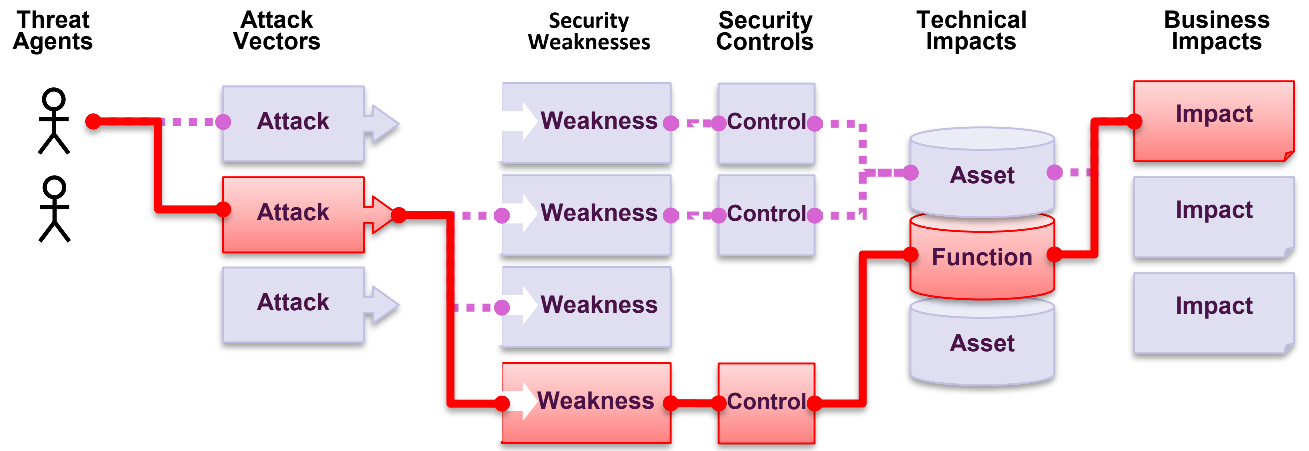
OWASP Top 10 - 2017		
1	Injection	
2	Broken Authentication	
3	Sensitive Data Exposure	
4	XML External Entities (XXE)	
5	Broken Access Control	
6	Security Misconfiguration	
7	Cross-Site Scripting (XSS)	
8	Insecure Deserialization	
9	Using Components with Known Vulnerabilities	
10	Insufficient Logging & Monitoring	

OWASP Top 10 - 2021	
1	Broken Access Control
2	Cryptographic Failure
3	Injection
4	Insecure Design
5	Security Misconfiguration
6	Vulnerable and Outdated Components
7	Identification and Authentication Failures
8	Software and Data Integrity Failures
9	Security Logging and Monitoring Failures
10	Server-Side Request Forgery

Which metrics can you extract from OWASP Top 10?

Observation:

- Attackers can potentially use many different paths through your application to do harm to your business or organization.



- Each of these paths represents a risk that may, or may not, be serious enough to warrant attention.

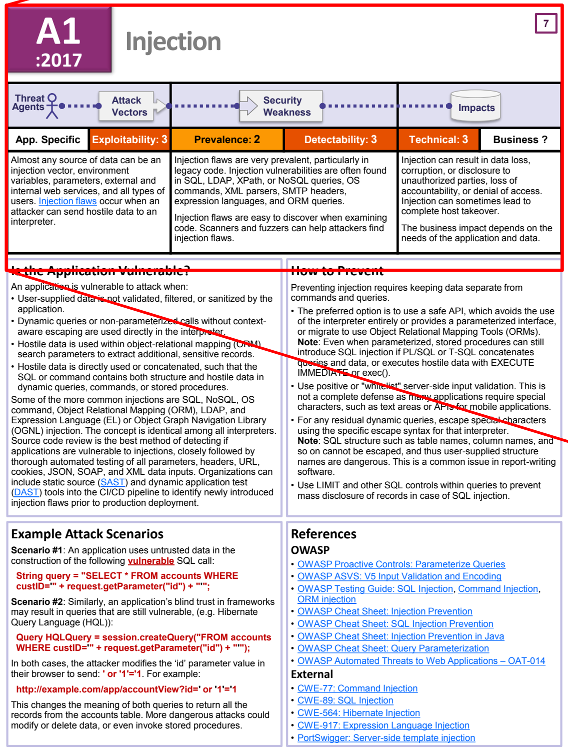
Source: OWASP Top 10 – 2017 The Ten Most Critical Web Application Security Risks, https://owasp.org/www-project-top-ten/2017/

How did OWASP rank threats?

For each of identified threats, OWASP provides generic information about likelihood and technical impact using the ratings scheme based on the OWASP Risk Rating Methodology

Threat Agents	Exploitability	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
Application Specific	Easy: 3	Widespread: 3	Easy: 3	Severe: 3	Business Specific
	Average: 2	Common: 2	Average: 2	Moderate: 2	
	Difficult: 1	Uncommon: 1	Difficult: 1	Minor: 1	

OWASP top 10 Example



A1 :2017

Injection

7

Threat Agents	Attack Vectors	Security Weakness		Impacts	
App. Specific	Exploitability: 3	Prevalence: 2	Detectability: 3	Technical: 3	Business ?
Almost any source of data can be an injection vector, environment variables, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile data to an interpreter.		Injection flaws are very prevalent, particularly in legacy code. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries. Injection flaws are easy to discover when examining code. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover. The business impact depends on the needs of the application and data.	

Is the Application Vulnerable?

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

How to Prevent

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).
  Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
  Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE
custID='" + request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g. Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts
WHERE custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in their browser to send: **' or '1'='1**. For example:

```
http://example.com/app/accountView?id=' or '1'='1
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data, or even invoke stored procedures.

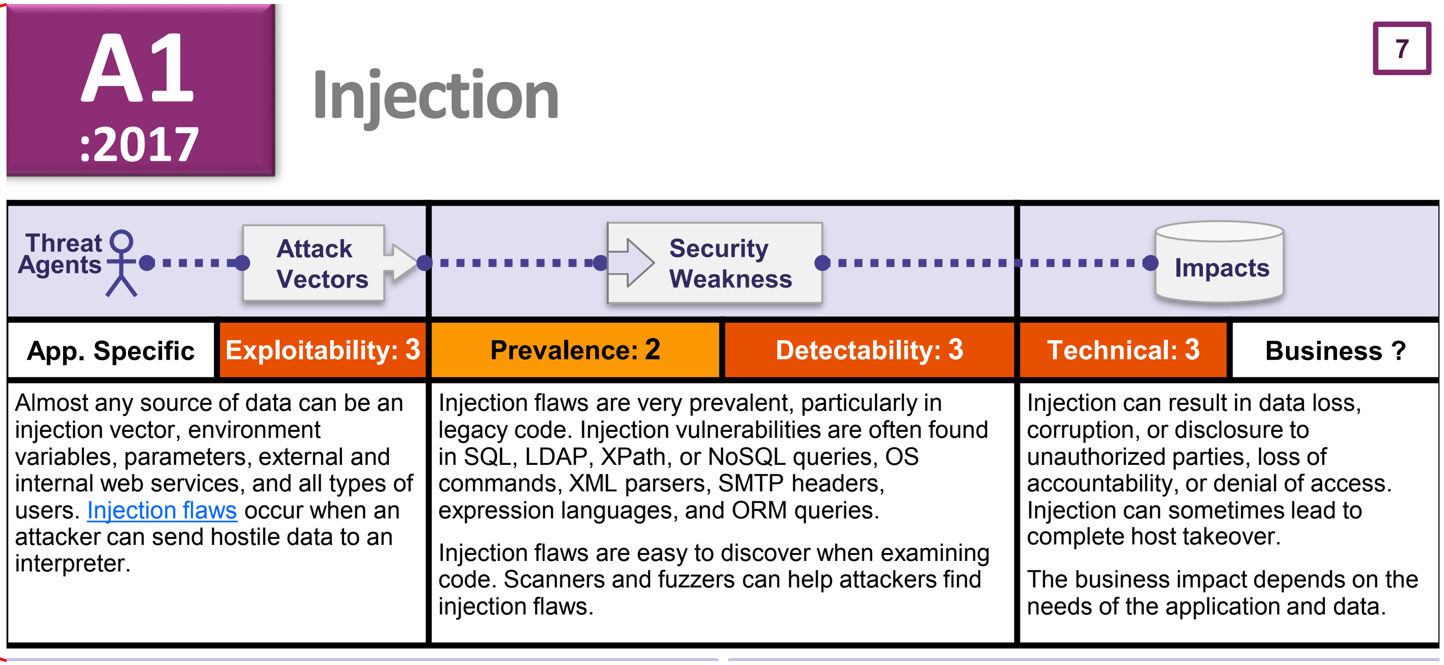
References

OWASP

- OWASP Proactive Controls: Parameterize Queries
- OWASP ASVS: V5 Input Validation and Encoding
- OWASP Testing Guide: SQL Injection, Command Injection, ORM injection
- OWASP Cheat Sheet: Injection Prevention
- OWASP Cheat Sheet: SQL Injection Prevention
- OWASP Cheat Sheet: Injection Prevention in Java
- OWASP Cheat Sheet: Query Parameterization
- OWASP Automated Threats to Web Applications – OAT-014

External

- CWE-77: Command Injection
- CWE-89: SQL Injection
- CWE-564: Hibernate Injection
- CWE-917: Expression Language Injection
- PortSwigger: Server-side template injection



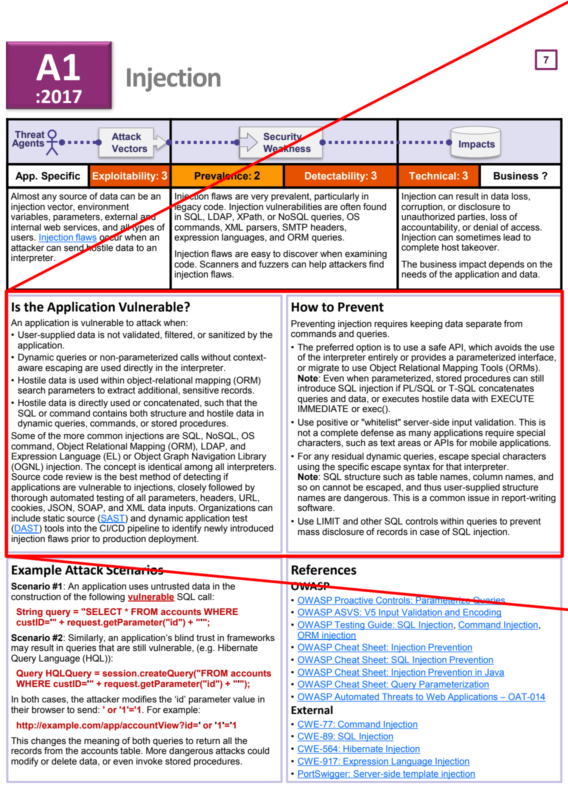
A1 :2017

Injection

7

Threat Agents	Attack Vectors	Security Weakness		Impacts	
App. Specific	Exploitability: 3	Prevalence: 2	Detectability: 3	Technical: 3	Business ?
Almost any source of data can be an injection vector, environment variables, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile data to an interpreter.		Injection flaws are very prevalent, particularly in legacy code. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries. Injection flaws are easy to discover when examining code. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover. The business impact depends on the needs of the application and data.	

OWASP top 10 Example



A1 :2017 Injection

Threat Agents → Attack Vectors → Security Weakness → Impacts

App. Specific	Exploitability: 3	Prevalence: 2	Detectability: 3	Technical: 3	Business ?
Almost any source of data can be an injection vector, environment variables, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile data to an interpreter.		Injection flaws are very prevalent, particularly in legacy code. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries. Injection flaws are easy to discover when examining code. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover. The business impact depends on the needs of the application and data.	

Is the Application Vulnerable?

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

How to Prevent

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).
  Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
  Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE
custID='" + request.getParameter("id") + "'";
```

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g. Hibernate Query Language (HQL)):

```
Query HQLQuery = session.createQuery("FROM accounts
WHERE custID='" + request.getParameter("id") + "'");
```

In both cases, the attacker modifies the 'id' parameter value in their browser to send: **' or '1'='1**. For example:

```
http://example.com/app/accountView?id=' or '1'='1
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data, or even invoke stored procedures.

References

OWASP

- OWASP Proactive Controls: Parameterize Queries
- OWASP ASVS: V5 Input Validation and Encoding
- OWASP Testing Guide: SQL Injection, Command Injection, ORM injection
- OWASP Cheat Sheet: Injection Prevention
- OWASP Cheat Sheet: SQL Injection Prevention
- OWASP Cheat Sheet: Injection Prevention in Java
- OWASP Cheat Sheet: Query Parameterization
- OWASP Automated Threats to Web Applications – OAT-014

External

- CWE-77: Command Injection
- CWE-89: SQL Injection
- CWE-564: Hibernate Injection
- CWE-917: Expression Language Injection
- PortSwigger: Server-side template injection

Is the Application Vulnerable?

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or stored procedures.

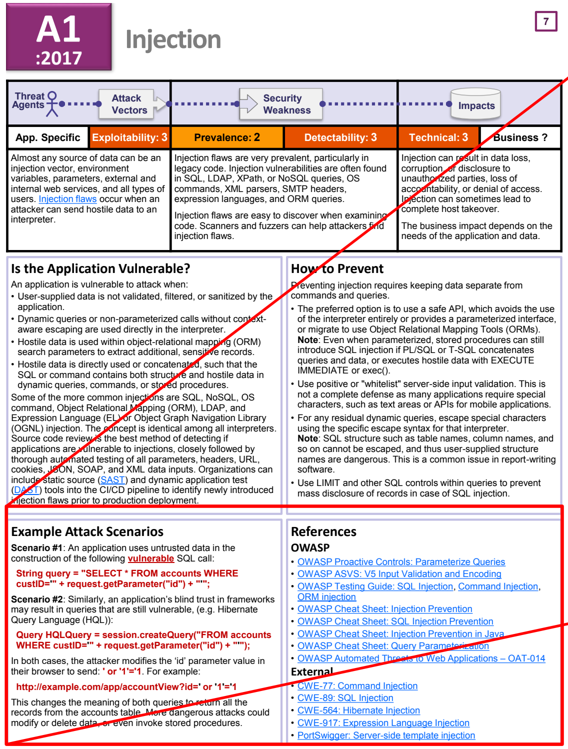
Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

How to Prevent

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs).
  Note: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter.
  Note: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

OWASP top 10 Example



A1 :2017 Injection

7

App. Specific	Exploitability: 3	Prevalence: 2	Detectability: 3	Technical: 3	Business ?
Almost any source of data can be an injection vector, environment variables, parameters, external and internal web services, and all types of users. Injection flaws occur when an attacker can send hostile data to an interpreter.		Injection flaws are very prevalent, particularly in legacy code. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries. Injection flaws are easy to discover when examining code. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover. The business impact depends on the needs of the application and data.	

Is the Application Vulnerable?

An application is vulnerable to attack when:

- User-supplied data is not validated, filtered, or sanitized by the application.
- Dynamic queries or non-parameterized calls without context-aware escaping are used directly in the interpreter.
- Hostile data is used within object-relational mapping (ORM) search parameters to extract additional, sensitive records.
- Hostile data is directly used or concatenated, such that the SQL or command contains both structure and hostile data in dynamic queries, commands, or stored procedures.

Some of the more common injections are SQL, NoSQL, OS command, Object Relational Mapping (ORM), LDAP, and Expression Language (EL) or Object Graph Navigation Library (OGNL) injection. The concept is identical among all interpreters. Source code review is the best method of detecting if applications are vulnerable to injections, closely followed by thorough automated testing of all parameters, headers, URL, cookies, JSON, SOAP, and XML data inputs. Organizations can include static source (SAST) and dynamic application test (DAST) tools into the CI/CD pipeline to identify newly introduced injection flaws prior to production deployment.

How to Prevent

Preventing injection requires keeping data separate from commands and queries.

- The preferred option is to use a safe API, which avoids the use of the interpreter entirely or provides a parameterized interface, or migrate to use Object Relational Mapping Tools (ORMs). **Note**: Even when parameterized, stored procedures can still introduce SQL injection if PL/SQL or T-SQL concatenates queries and data, or executes hostile data with EXECUTE IMMEDIATE or exec().
- Use positive or "whitelist" server-side input validation. This is not a complete defense as many applications require special characters, such as text areas or APIs for mobile applications.
- For any residual dynamic queries, escape special characters using the specific escape syntax for that interpreter. **Note**: SQL structure such as table names, column names, and so on cannot be escaped, and thus user-supplied structure names are dangerous. This is a common issue in report-writing software.
- Use LIMIT and other SQL controls within queries to prevent mass disclosure of records in case of SQL injection.

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following **vulnerable** SQL call:

String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") + "'";

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g. Hibernate Query Language (HQL)):

Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");

In both cases, the attacker modifies the 'id' parameter value in their browser to send: **' or '1'='1**. For example:

http://example.com/app/accountView?id=' or '1'='1

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data, or even invoke stored procedures.

References

OWASP

- OWASP Proactive Controls: Parameterize Queries
- OWASP ASVS: V5 Input Validation and Encoding
- OWASP Testing Guide: SQL Injection, Command Injection, ORM injection
- OWASP Cheat Sheet: Injection Prevention
- OWASP Cheat Sheet: SQL Injection Prevention
- OWASP Cheat Sheet: Injection Prevention in Java
- OWASP Cheat Sheet: Query Parameterization
- OWASP Automated Threats to Web Applications – OAT-014

External

- CWE-77: Command Injection
- CWE-89: SQL Injection
- CWE-564: Hibernate Injection
- CWE-917: Expression Language Injection
- PortSwigger: Server-side template injection

Example Attack Scenarios

Scenario #1: An application uses untrusted data in the construction of the following **vulnerable** SQL call:

String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") + "'";

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g. Hibernate Query Language (HQL)):

Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");

In both cases, the attacker modifies the 'id' parameter value in their browser to send: **' or '1'='1**. For example:

http://example.com/app/accountView?id=' or '1'='1

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify or delete data, or even invoke stored procedures.

References

OWASP

- OWASP Proactive Controls: Parameterize Queries
- OWASP ASVS: V5 Input Validation and Encoding
- OWASP Testing Guide: SQL Injection, Command Injection, ORM injection
- OWASP Cheat Sheet: Injection Prevention
- OWASP Cheat Sheet: SQL Injection Prevention
- OWASP Cheat Sheet: Injection Prevention in Java
- OWASP Cheat Sheet: Query Parameterization
- OWASP Automated Threats to Web Applications – OAT-014

External

- CWE-77: Command Injection
- CWE-89: SQL Injection
- CWE-564: Hibernate Injection
- CWE-917: Expression Language Injection
- PortSwigger: Server-side template injection