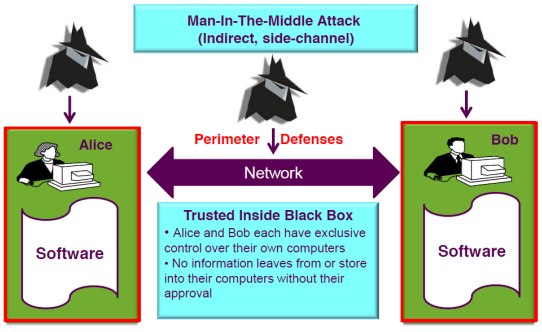
Software Protection

Ci mettiamo ora nell'ottica, in cui **l'asset da proteggere sia il codice** → l'evoluzione dei dispositivi, ha reso quest'ultimi sempre più aperti, ovverosia sono dispositivi con cui è sempre più facile interagire e questo naturalmente la superficie di attacco e può favorire intromissioni da parte degli attaccanti.

Lo scenario tipico della crittografia viene detto **Black Box Attack** oppure **Grey Box Attack** → in questo scenario abbiamo due utenti (nell'immagine sotto: Alice e Bob), che sono entrambi utenti fidati e di conseguenza i computer di entrambi gli utenti viene considerato sicuro. Quello, però, che non è sicura è la comunicazione tra i due utenti → questo ci fa capire, che i due computer degli utenti si comportano come delle Black Box per l'attaccante, ovverosia l'attaccante non vede come si comportano i dispositivi degli utenti, bensì riesce solamente a vedere i messaggi che si scambiano. É proprio qui, quindi, che l'attaccante può attaccare ed è per questo che l'attaccante viene anche definito **Man-In-The-Middle** → in questo scenario, quindi, ci fidiamo dell'ambiente in cui viene eseguito il codice e dobbiamo controllare solamente la comunicazione.

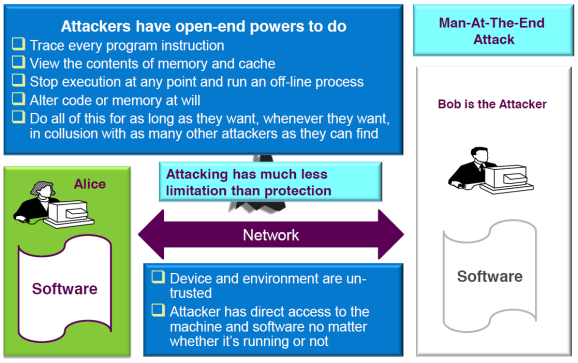


L'attaccante, quindi, si mette in mezzo alla comunicazione ed in particolare, possiamo avere:

- **Black Box Attack** → in cui l'attaccante non ha alcuna informazione su come vengono generati i messaggi scambiati durante la comunicazione tra i due utenti → chiaramente questo rende difficile l'azione dell'attaccante, in quanto per comprendere il funzionamento del software può solamente osservare i messaggi generati;
- **Grey Box Attack** → in cui l'attaccante ha alcune/qualche informazioni su come vengono generati i messaggi scambiati durante la comunicazione tra i due utenti.

Nello scenario **White Box**, siamo completamente all'opposto rispetto allo scenario Black Box. Nello scenario White Box, abbiamo sempre un utente fidato (nell'immagine sotto Alice), mentre l'altro, non rappresenta un interlocutore fidato (nell'immagine sotto Bob) e di conseguenza potrebbe essere l'attaccante → in questo scenario, chiaramente Bob ha accesso a tutte le informazioni presenti sulla macchina in cui si trova e quindi lo scenario di attacco prende il nome di **Man-At-The-End**, in quanto l'attaccante non si pone nel mezzo, bensì ha controllo della macchina e di tutte le applicazioni che vi girano sopra → quindi, l'attaccante ha pieno potere e quindi non c'è limite a quello che l'attacante può fare, ovvero:

- oltre ad usare tutti i tool di disassembly, debugging e decompilazione;
- l'attaccante può anche prendere versioni successive del codice ed analizzarlo, ad esempio capendo vulnerabilità delle vecchie versioni del codice e attaccando gli utenti che non hanno aggiornato il codice.



Chiaramente, se vogliamo garantire la sicurezza, più si schiarisce la Box (quindi più andiamo verso una White Box) la sicurezza diventa più difficile, in quanto nello scenario Black Box l'attaccante vede solamente il comportamento input/output e tutto il resto lo deve ricavare. Più la Box da nera diventa trasparente e fa vedere tutto ciò che accade, più l'attaccante ha possibilità di imparare informazioni sul funzionamento del software e quindi diventerà sempre più difficile proteggere quest'ultimo → in questo senso, vi è la White-Box Security, ovverosia una tecnica di crittografia che cerca di ottenere la sicurezza Black Box in uno scenario White Box.

Quando parliamo di software protection, quindi, siamo in uno scenario Man-At-The-End, ovverosia in uno scenario in cui non ci fidiamo dell'ambiente in cui il codice viene eseguito.
La software protection, cerca, allora di capire come possiamo difenderci dall'attaccante nello scenario White Box, in cui abbiamo detto che quest'ultimo può fare tutto quello che vuole e quindi può utilizzare tutti i tool per imparare e capire come funziona il nostro software.

Ci stiamo, quindi, immaginando una situazione, in cui l'attaccante prende un codice proprietario e lo manometta, al fine di ottenere un qualche tipo di profitto per l'attaccante e un danno per l'utente. Le tre fasi principali di un attacco al software (da parte dell'attaccante) sono:

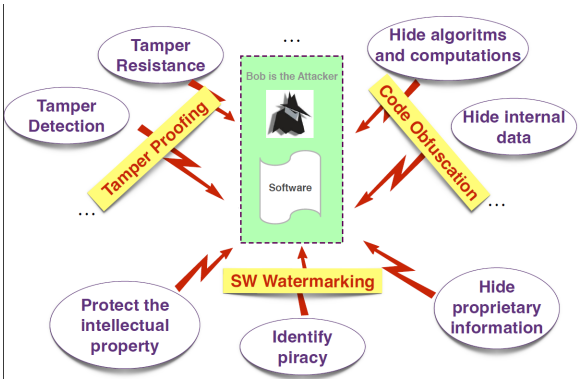
1. **Reverse engineering** → prima fase fondamentale è quella del reverse engineering, in quanto spesso l'attacante non è in possesso del codice sorgente, bensì è in possesso solamente dell'eseguibile e quindi deve capirne il suo funzionamento. In questa prima fase vi si collocano, quindi, tutte le tecniche di reverse engineering per capire il funzionamento interno del codice (ovvero: cosa fanno le varie parti; chi sono i responsabili delle azioni, ecc..)
2. **Tampering** → una volta che l'attaccante ha capito il funzionamento del codice, quest'ultimo può decidere che manomissione adoperare e quindi andrà a:
    a. cambiare dei dati;
    b. mettere delle chiavi;
    c. cancellare delle porzioni di codice;
    d. cambiare delle porzioni di codice per sviluppare, ad esempio, un programma concorrente.
3. **Profit** → una volta che l'attaccante ha manomesso il software, il suo obiettivo è quello di ottenere un profitto economico.

Le tecniche di protezione del software, quindi, vorrebbero garantire di:

- rendere più complicata la fase di reverse engineering → quindi di andare, in un qualche modo, ad impoverire i risultati dei tool di analisi statica o dinamica, che possono essere utilizzati a supporto della comprensione del codice → quindi vorremmo, che il disassemblaggio o la decompilazione del codice producessero un codice che crasha o che in alcuni punti non si comporti come il codice originale → vorremmo, quindi, complicare il compito dell'attaccante;

- resistere al tampering → resistere al tampering non tanto dal punto di non permettere le modifiche del codice, bensì intendiamo:
  - rendere più difficile capire dove effettuare le modifiche;
  - utilizzare dei tool di tampering detection, che mi permettono di fare l'hash di alcune porzioni di codice e di capire se il codice è stato manomesso ed eventualmente evitare che il codice manomesso venga eseguito.
- resistere alla clonazione e alla distribuzione delle copie non autorizzate del codice;
- mantenere i segreti.

Visivamente, quindi, possiamo immaginare diverse tipologie di attacchi al software:



Per esempio, gli attacchi in alto a destra (Hide algorithms and computations e Hide Internal data) che mirano a trovare dei segreti all'interno del programma, dove per segreti intendiamo:

- chiavi;
- particolari strategie implementate dall'algoritmo, che quindi rappresentano il valore dell'algoritmo.

Oppure, abbiamo gli attacchi che mirano a ledere la proprietà intellettuale, che mirano quindi ad impedire l'utilizzo di porzioni di codice: Infine, abbiamo gli attacchi che vanno direttamente a manomettere il codice.

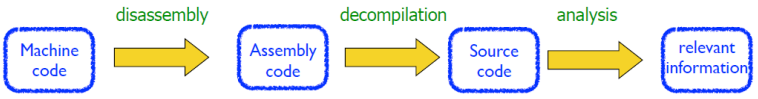
Nell'immagine sopra, possiamo vedere che gli attacchi sono stati raggruppati in tre macro-categorie → questo perchè, sono state sviluppate tre famiglie di **tecniche software** per difendersi da queste tipologie di attacchi ed in particolar modo per **difendere la proprietà intellettuale** → da notare, che abbiamo detto che sono tecniche software, ovverosia sono tecniche che vanno a trasformare il codice, in

modo da proteggerlo (quindi non sono mezzi legali o hardware). Queste tre tecniche software sono:

1. **Code Obfuscation** → l'offuscamento del codice è il processo di modifica di un eseguibile, in modo tale che quest'ultimo non sia più utile all'attaccante, ma che rimanga **completamente funzionante**. Il processo può modificare le istruzioni del metodo o i metadati, ma **non altera l'output** del programma, ovverosia l'offuscamento del codice non deve alterare in alcun modo la semantica del programma → l'offuscamento del codice, quindi, rende il programma più difficile da analizzare.
2. **SW Watermarking** → tecnica che vuole riconoscere le copie piratate (quindi non autorizzate) del codice, inserendo all'interno di quest'ultimo una firma.
3. **Tamper Detection** → tecnica che vuole riconoscere quando il codice è stato manomesso e tipicamente lanciare una qualche forma di allarme. Tale manomissione, successivamente, verrà gestita dal team di response.

Quindi, quando si parla di protezione del software, il nostro obiettivo è quello di proteggere il software, in quanto all'interno dell'oggetto software stesso risiede il valore. Tipicamente le fasi per attaccare un programma software sono (come abbiamo detto prima):

- comprenderne il suo comportamento → quindi si ha una prima fase di reverse engineering. Questa fase di reverse engineering parte dall'eseguibile e cerca di ricostruire il codice ad alto livello, in quanto quest'ultimo è più semplice da capire ed analizzare, al fine da estrarre l'informazione rilevante.



Sorge spontanea una domanda: "**A che livello devo applicare le tecniche di protezione?**" In un qualche modo, la protezione serve a tutti i livelli e quindi vorremmo impedire sia il disassembly, sia la decompilazione e naturalmente vorremmo impedire l'analisi. In realtà tante delle tecniche di offuscamenti del codice si concentrano sulla fase di decompilazione.

- una volta che abbiamo capito il suo funzionamento, possiamo decidere come utilizzare i pezzi di codice che lo compongono, ovverosia possiamo decidere se:
  - riutilizzare delle porzioni di codice;
  - oppure andare a modificarle.

> 💡 Nonostante tutte le tecniche possibili che possiamo adottare, con tempo, impegno e determinazione sufficienti, un programmatore competente può **sempre** fare reverse engineering di qualsiasi applicazione → quindi il nostro obiettivo è quello di rallentare l'attaccante e rendere il reverse engineering talmente costoso che per l'attaccante non è più conveniente.

Nella fase di reverse engineering, possiamo misurare l'efficacia dell'offuscamento andando a guardare le risposte, che i tool di analisi del codice ci danno prima e dopo l'offuscamento → capiamo, allora, che è difficile comprendere quanto l'offuscamento complichi il lavoro dell'attaccante umano, dato che facciamo fatica a fornire un modello di attaccante → è chiaro, che se complichiamo tutte le risposte, che i tool di analisi forniscono, allora avrò certamente complicato il lavoro dell'attaccante umano. Quindi, **gli attaccanti ce li possiamo immaginare come degli analizzatori statici o dinamici, che vogliono estrarre informazioni dal codice**. In particolare, abbiamo che:

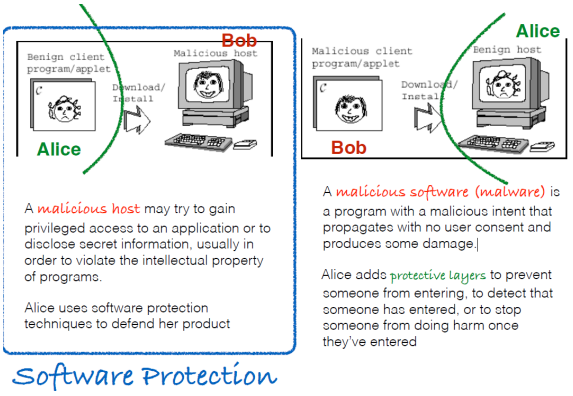
- **analisi statica** → essa analizza il codice senza eseguirlo. Siccome, quindi, l'analisi statica non esegue il codice, essa analizza il codice e considera tutte le possibili cose che potrebbero succedere durante l'esecuzione → si dice, tipicamente, che l'analisi statica è conservativa, ovverosia vede come possibili comportamenti tutti i comportamenti che accadranno nella realtà più gli altri che non accadranno, ma che l'analisi vede come possibili → un esempio è quando vi è un if, il quale (per la sua condizione) entra sempre nel ramo vero. L'analisi statica considera entrambi i cammini, anche se quello falso non accadrà mai → questo comporta il fatto, che se l'analisi statica non trova errori nel codice, sicuramente non ci saranno errori. Se invece trova un errore, allora bisogna capire se l'errore si trova nei casi possibili oppure in quelli che non si verificheranno mai.
- **analisi dinamica** → essa prende in input il programma e un campione di dati ed analizza il programma eseguendolo sull'insieme di dati in input considerato → il limite dell'analisi dinamica è che non posso provare tutte le tracce, ovverosia non posso provare tutte le possibili esecuzioni e quindi l'analisi dinamica mi fornisce un sottoinsieme delle tracce → questo ci fa capire, che se con l'analisi dinamica trovo un errore, allora siamo sicuri c'è, mentre se non lo trovo, possiamo aver considerato solamente le tracce sicure e non quelle non sicure.

> 💡 Il nostro obiettivo, quindi, è che entrambe le tipologie di analisi forniscano dei risultati degradati (ovvero meno precisi) per l'attaccante.

Per proteggerci dagli attaccanti, quindi, possiamo ricorrere a:

- misure legali;
- le tre tecniche software nominate precedentemente, ovvero:
  - Code Obfuscation;
  - SW Watermarking;
  - Tamper Detection.

In particolar modo, quando parliamo di software protection ci riferiamo al fatto, che un host malintenzionato potrebbe tentare di ottenere l'accesso privilegiato a un'applicazione o ad informazioni segrete, solitamente per violare la proprietà intellettuale dei programmi → quindi, quando parliamo di software protection, abbiamo che il programma è fidato e sicuro, mentre l'host è malevolo. Caso opposto, invece, è quando si entra nel campo dei malware → un software dannoso (malware) è un programma con un intento malevolo, che si propaga senza il consenso dell'utente e produrre un danno. In questo caso, quindi, il programma è malevolo, mentre l'host è sicuro.



Quando pensiamo alla protezione del codice, allora, sorge spontanea la domanda: "**Perchè non utilizziamo delle tecniche di crittografia?**" Per il semplice fatto, che il codice criptato non può essere eseguito e quindi, anche se avessimo delle porzioni di codice cifrato, esso ad un certo punto dovrà essere decifrato per essere eseguito e questa decifratura dovrà essere fatta sulla macchina dell'utente → ma visto che,

nella software protection si assume che l'attaccante abbia il controllo della macchina, allora l'attaccante potrebbe vedere il codice in chiaro e quindi la crittografia non è una possibili scelta da adottare.

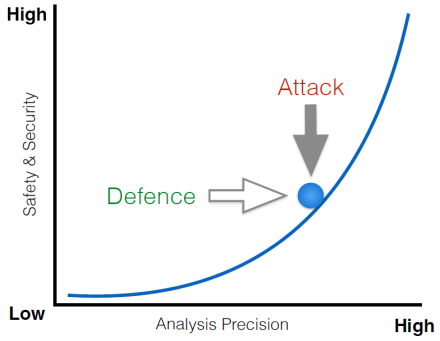
Vediamo, a questo punto, come possiamo complicare la fase di Reverse Engineering:

- La prima possibilità (oltre all'offuscamento) è di utilizzare dei **dispositivi hardware** e quindi legare la corretta esecuzione del programma alla presenza di un dispositivo hardware → un tipico esempio di dispositivo hardware è il **dongle**, ovvero un piccolo dispositivo hardware che si collega alla porta seriale o USB di un computer → questa prima soluzione **limita la portabilità**;
- Un'altra soluzione è l'**Encryption** → immaginiamoci, quindi, che vi sia il Server (che è fidato) e il Client (che non è fidato), dove gira il codice che vogliamo proteggere. Se nel Client vi è l'Encrypted Code, ovverosia il codice cifrato, succede (come abbiamo detto prima) che ad un certo punto tale codice cifrato deve essere decifrato e di conseguenza l'attaccante (che ha il controllo del Client) può prendere il codice non cifrato ed analizzarlo → vi è quindi un problema di un **basso livello di sicurezza**;
- Un'altra soluzione possibile è quella della **Remote Execution** → questa soluzione, prevede il fatto, che visto che il Client non è fidato e sicuro, allora ad esso non gli viene fornito il codice, bensì al Client gli vengono forniti dei servizi → questa soluzione ha certamente come vantaggio un alto livello di sicurezza, però ha come svantaggio un elevata pesantezza della comunicazione Client-Server;
- Un'altra soluzione è di combinare le due soluzioni precedenti e di ottenere la **Partial Remote Execution** → questa soluzione prevede, che le parti che non devono essere assolutamente manomesse vengono mantenute sul Server (e quindi in un ambiente fidato e sicuro) e sulle applicazioni Client viene distribuito il codice non critico, ovverosia codice che non contiene informazioni sensibili o chiavi → l'attaccante, quindi, può decompilare questo codice non critico, ma non produce alcun danno;
- L'ultima soluzione possibile è la **Full Code Deployed** → soluzione che prevede di offuscare il codice e di conseguenza l'attaccante può procedere in due modi:
  - de-offuscare il codice;
  - oppure deve analizzare il codice offuscato.

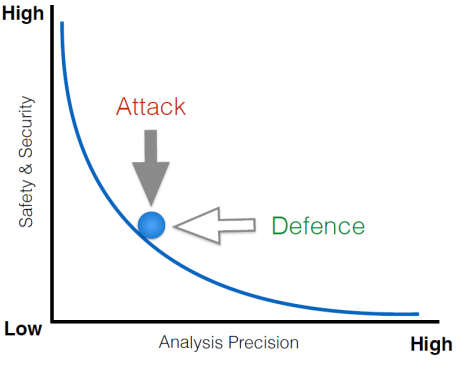
Vi sono diversi metodi ci offuscamento del codice:

- Renaming;
- Encryption;
- Control flow;
- Data flow;
- Virtualisation.

In generale, l'offuscamento ci può difendere dai tool di analisi automatica e complica l'analisi del codice da parte degli attaccanti, ma non può difenderci da tutti gli attacchi. Da notare anche un altro aspetto, ovverosia quando facciamo Code Debugging, il nostro obiettivo è di cercare ed eliminare tutte le vulnerabilità, al fine di rendere il software più sicuro → vogliamo, quindi, migliorare la precisione dell'analisi e conseguentemente andiamo anche a migliorare la sicurezza del software. Questo lo si può osservare dalla seguente figura:



Nella software protection, invece, andiamo a ribaltare la curva. Questo perchè, per difendere il software dobbiamo degradare la precisione dell'analisi del codice → quindi voglio rendere l'analisi meno precisa e al diminuire della precisione dell'analisi ottengo un aumento della sicurezza del software. Questo lo si può osservare nella seguente figura:



Dopo aver analizzato le due curve sopra, riportiamo all'attenzione un concetto detto precedentemente, ovvero: Se noi ci immaginiamo un intervallo temporale in cui siamo esposti ad un attacco, l'attaccante sarà disposto a rubare informazioni critiche dell'utente, solamente se con un investimento (relativamente) basso riesce ad ottenere un guadagno economico (relativamente) alto → se attraverso i meccanismi di sicurezza, noi riusciamo a far alzare l'investimento all'attaccante (dato che abbiamo detto che è impossibile rendere il codice non analizzabile) e conseguentemente l'investimento è più alto del guadagno, l'attaccante non avrà più interesse nel rubare le nostre informazioni critiche e di conseguenza ci siamo difesi dall'attacco.

Diamo ora una panoramica sulle tre tecniche software per proteggerci dagli attacchi:

- **Code Obfuscation** → l'offuscamento del codice viene definito per la prima volta da Christian Collbert nel 1998. In particolare, Collbert definisce l'offuscamento come una trasformazione dei programmi, in quanto effettivamente trasforma un programma che calcola una cosa, in un altro programma che calcola esattamente la stessa cosa dell'altro, ma in modo più complicato → quindi l'offucamento preserva il **comportamento osservazionale** del programma ed è **potente**, ovverosia rende il programma più difficile da analizzare. L'offuscamento del codice viene valutato in base a quattro parametri:
  - **potenza** → quanto è più difficile per l'attaccante analizzare il programma offuscato rispetto al programma originale?
  - **resistenza** → quanto è difficile eliminare l'offuscamento dal programma offuscato?
  - **costo** → quanto overhead computazionale (in termini di tempo e spazio) l'offuscamento aggiunge al programma?
  - **invisibilità** → quanto il codice offuscato si adatta al codice originale?

  La bontà dell'offuscamento, quindi, è data dalla somma di questi quattro parametri.
- **SW Watermarking** → esso ha a che fare con la pirateria del software, in cui si ha un utente malintenzionato che compra un programma e lo vende come suo e di conseguenza mette in circolazione delle copie illecite del programma. Con il Watermarking voglio andare, in un qualche modo, a nascondere all'interno del software un'informazione che indichi la proprietà → il Watermarking non è una tecnica di protezione preventiva, ovvero non previene che qualcun altro possa

utilizzare il programma, bensì mi permette di riconoscere se è avvenuto un riutilizzo illecito. Il Watermarking deve essere:

- **resistente** → ovvero se viene offuscato deve comunque rimanere e quindi deve essere resistente alle tecniche di offuscamento del codice, che ne mantengono la semantica;
- **invisibile**;
- **high-bit rate** → ovvero devo avere la possibilità di nascondere più di un bit di informazione, ma almeno una sequenza di bit;
- **basso costo** → in particolar modo, intendiamo che deve avere una bassa degradazione delle performance (in termini di tempo e spazio) dell'esecuzione del programma watermarcato rispetto al programma originale.

(Il Watermarking spesso viene paragonato alle tecniche di stenografia, ma in realtà non ha nulla a che fare).

Il Watermarking funziona nel seguente modo: gli algoritmi di Watermarking devono avere due processi:

1. Processo di **Embending** → prende il programma ed inserisca la firma ed una chiave ed infine restituisce il programma marcato;
2. Processo di **Extraction** → prende il programma marcato e con la chiave restituisce la firma.

- **SW Tampering** → vorrebbe garantire che il codice in esecuzione sia il codice che è stato sviluppato dall'utente che l'ha pensato e quindi che il codice non sia stato manomesso o modificato. Tipicamente gli algoritmi di tampering si compongono da due parti:
  - Una parte di **check** → in cui si monitora lo "stato di salute" del sistema, andando a testare un insieme di invarianti e restituisce TRUE se non viene trovato nulla di sospetto → in un certo senso, questa parte verifica l'integrità del software. Quello che vado a checkare può essere:
    - il codice → andando a calcolare le hash di alcune porzioni di codice;
    - lo stato di esecuzione della macchina → quindi se ad esempio, il sistema mi risponde TRUE vuol dire che non è stato manomesso il sistema;
    - sull'ambiente di esecuzione del programma.

- Una parte di **respond** → interroga la parte di check per verificare se il programma stia funzionando come previsto e in caso contrario emette una risposta di manomissione, come ad esempio:
  - terminando il programma;
  - ripristinare il programma (facendo quindi delle patch al codice);
  - segnalare l’attacco;
  - punire l’attaccante (distruggendo, ad esempio, il codice).