Software Tampering

Abbiamo detto, che per la protezione del codice abbiamo tre possibile strade da percorrere, ovvero:

- offuscamento;
- tampering;
- watermarking.

Se l’offuscamento del codice vuole prevenire la **manomissione del codice, andando a rendere più difficile la comprensione di quest’ultimo per l’attaccante,** e quindi l’asset che vuole proteggere l’offuscamento è il codice, il tampering, invece, assicurare che i programmi vengano eseguiti come previsto, anche in presenza di un un attaccante, che tenta di interrompere, monitorare o modificare l'esecuzione. Un algoritmo di **tamper-detection**, quindi:

- rende difficile la manomissione;
- rileva quando si è verificata la manomissione e/o una modifica del codice non autorizzata e quindi in questo caso si parla di **tamper-detection**;
- risponde all'attacco.

Gli algoritmi di tamper-detection, quindi, prevedono:

- una fase di **detection** → ovvero una fase in cui si cerca di accorgersi se il codice è stato manomesso e quindi si tratta di una fase in cui si rileva quando si è verificata la manomissione;
- una fase di **respond** → dopo che si che individuata la manomissione, si dovrà rispondere a tale manomissione, andando ad esempio a:
    - abortire il processo;
    - ripristinare una copia integra del codice.

Un classico esempio di tamper-detection, è quando si vuole andare a proteggere la manomissione dei controlli di licenza, in modo tale che features a pagamento o prodotti a pagamento non vengano resi disponibile senza il pagamento della corrispondente somma di denaro.

A questo punto, vediamo quali sono i possibili scenari di attacco, ovverosia vediamo quali azioni può compiere l’attaccante. Abbiamo che l’attaccante può:

- modificare il **programma** → in quanto, l'attaccante ha come scopo quello di far fare al programma qualcosa di diverso rispetto a quello che farebbe normalmente durante la sua esecuzione originale;
- modificare l'**ambiente** in cui il programma viene eseguito;
- modificare le **librerie** utilizzate dal programma.

> 💡 Il concetto, quindi, che dobbiamo cogliere è che la manomissione non riguarda solamente/esclusivamente il programma e di conseguenza, dovremmo cercare di controllare sia il programma sia le componenti associate al programma stesso (quali ad esempio: gli input, le librerie utilizzate e l'ambiente di esecuzione) e l'obiettivo finale sarebbe quello di garantire, che tutto quello che riguarda l'esecuzione del programma sia "sano", ovverosia non corrotto/manomesso.

Per riuscire a garantire ciò, quello che si può andare a fare è:

- **Code checking** → chiaramente, il modo più semplice di fare Code checking è attraverso l'hash e di conseguenza, esso (ovvero il code checking) è molto suscettibile a modifiche, dato che ogni volta che viene modificato il codice, l'hash è diversa;
- **Result checking** → verificare che il risultato del calcolo sia corretto, attraverso dei meccanismi di challenge-response (questo perchè se ad una challenge, il programma risponde con un risultato a noi aspettato (ovvero con un risultato che ci aspettiamo), allora siamo in grado di capire che il programma non è stato manomesso). Controllare la validità del risultato del calcolo è spesso computazionalmente più economico dell'esecuzione del calcolo stesso;
- **Environment checking** → la cosa più difficile da controllare per un programma è la validità del suo ambiente di esecuzione, dato che l'esecuzione può avvenire sotto un emulatore, ma soprattutto perchè l'ambiente può variare in base Sistema operativo utilizzato.

> 💡 Chiaramente, un meccanismo di tamper-detection è migliore quanto prima è in grado di individuare la manomissione. Idealmente, quindi, vorremmo che la manomissione venga individuata prima che il codice manomesso venga eseguito.

Una volta individuata la manomissione, possiamo:

- terminare il programma;
- ripristinare il programma al suo stato corretto, patchando il codice codice manomesso;
- restituire deliberatamente risultati non corretti, che magari si deteriorano lentamente nel tempo;
- ridurre le prestazioni del programma;
- segnalare l'attacco andando ad avvertire la vittima;
- punire l'attaccante distruggendo:
  - il programma o gli oggetti nel suo ambiente;
  - cancellare la directory home;
  - distruggere il computer flashando ripetutamente la memoria flash del bootloader.

Introspection

Vi è una prima famiglia di approcci al tampering, che lavora per **Introspection,** ovverosia il codice guarda sè stesso, nel tentativo di capire se è stato manomesso o meno (da qui deriva il termine Introspection) → abbiamo, quindi che il codice originale viene aumentato, ovverosia all'interno del codice originale viene inserita una funzione che va a calcolare l'hash di una porzione di codice, la quale (ovvero la porzione di codice) viene individuata da un indirizzo di partenza (start_address) e da un indirizzo finale (end_address), e successivamente andrà a verificare, che il valore di hash sia lo stesso di quello del codice non modificato. Vediamo questo concetto attraverso un esempio:

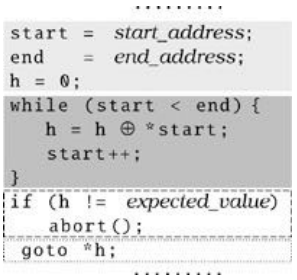


Figura 5

Vediamo, che abbiamo uno start_address ed un end_address e l'hash inizialmente è posta a zero. Successivamente, per tutte le istruzioni che si trovano tra lo

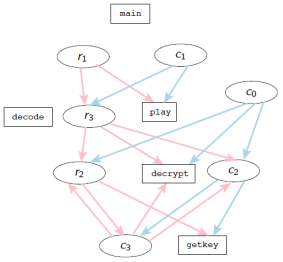
start_address e l'end_address andremo ad aggiornare il valore di hash. Una volta terminato di calcolare l'hash della porzione di codice, andiamo ad effettuare un controllo, in modo tale da verificare se il valore di hash ottenuto è uguale o diverso rispetto al valore aspettato. Se il valore è diverso, allora andiamo ad abortire il processo, mentre se il valore è uguale continuiamo con l'esecuzione del programma → vi sono quindi dei **checkers**, ovverosia delle funzioni che calcolano l'hash su una porzione di codice e il valore dell'hash lo confrontano con il valore atteso. Da notare, che abbiamo detto che i checkers controllano il codice, ma abbiamo anche che:

- alcuni checkers sono adibiti a controllarsi a vicenda, ovvero vi sono dei checkers che controllano gli altri checkers;
- alcuni checkers sono adibiti alla funzione di repair (ovvero di riparazione). Immaginiamoci, per esempio che il valore dell'hash sia diverso rispetto al valore originale. Tali checkers, quindi, vanno a sovrascrivere il codice con quello originale, in modo tale da eliminare il tampering (ovvero vanno a sovrascrivere la funzione manipolata con la funzione originale). Questo concetto lo possiamo osservare meglio con il seguente esempio:

Possiamo vedere, che se il valore di hash della funzione A è diverso al valore atteso, allora andiamo a sovrascrivere in memoria, il valore della funzione A con il valore originale della funzione A (ovvero A_COPY). Stessa cosa viene fatta anche con la funzione B.

Si va, quindi, a formare un **guard graph**, ovverosia un grafo formato da:
- **checkers;**
- **respond;**
- **funzioni originali del programma.**

Un esempio di guard graph lo si può vedere nell'immagine sotto:



Chiaramente l'inserimento dei checkers e delle responde vanno ad appesantire il codice e di conseguenza tali funzioni di check e di responde devono essere:

- **efficienti**;
- **stealthy**, nel senso che vorremmo che non fossero facilmente identificabili dall'occhio umano o dagli algoritmi di pattern-matching → per riuscire a fare questo, solitamente si vanno ad offuscare lo start_address e l'end_address (definendoli attraverso una routine dinamica) e a diversificare le funzioni di hash, in modo tale che se l'attaccante riconosce una funzione di hash, non vada a fare pattern-matching per identificare tutte le altre funzioni di hash presenti lungo il codice. Alcuni esempi di variazione di funzioni di hash sono:
  - riscrivere un ciclo FOR, come un ciclo WHILE infinito (quindi while(1)) con una return all'interno;
  - non calcolare l'hash di tutte le istruzioni, bensì calcolare l'hash solamente di alcune istruzioni (ovvero si parte da un indirizzo di base e a quest'ultimo di aggiunge uno step. Aggiungendo lo step, si arriva ad un'istruzione e di questa si calcola l'hash e poi dall'indirizzo di questa istruzione si aggiunge nuovamente lo step e così via);
  - utilizzare dei valori randomici (ovvero casuali) durante il calcolo della funzione di hash.

  Un altro punto da sottolineare per quanto riguarda la stealthiness, è che il confronto tra il valore della funzione di hash e il valore atteso è difficilmente un confronto stealthy per l'osservatore umano, dato che è un confronto tra un valore di una funzione e un valore costante (e solitamente gli IF non si fanno mai su valori costanti di questo tipo). Una possibile soluzione, consiste nel nascondere i letterali del valore atteso, che a meno che il codice non sia stato violato, abbia sempre valore zero (ovvero il valore atteso è sempre zero, a meno che il codice non sia stato violato). A questo scopo, possiamo utilizzare uno slot vuoto all'interno della regione da proteggere e successivamente assegnare a questo

slot, un valore che renda l'hash della regione pari a zero → tale slot prendono il nome di **Corrector Slot**.

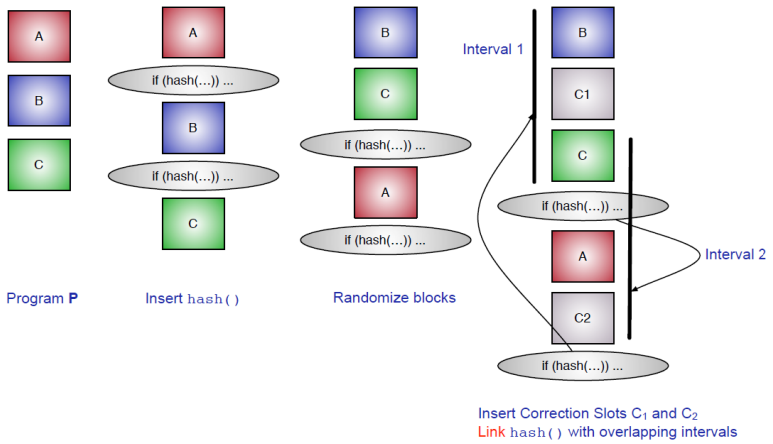
Naturalmente, inoltre, vi saranno dei **vincoli di dipendenza** su come inserire i checkers e le respond. In particolare i vincoli di dipendenza sono i seguenti:

- tra l'esecuzione di un checker e l'esecuzione della zona controllata dal checker, vi deve essere una respond (nell'eventualità in cui il checker vada a falso);
- tutte le respond devono essere controllare attraverso i checkers, dato che dobbiamo assicurarsi che le respond (le quali ricordiamoci vanno a sovrascrivere il codice) siano "sane".

Apriamo anche una breve parentesi su una tecnica di protezione del codice, che utilizza il concetto dei Corrector Slots. L'idea alla base di questa tecnica è la seguente: prendere il programma P che si vuole proteggere e suddividerlo in blocchi. Tra questi blocchi, poi, si vanno ad inserire delle funzioni di hash e successivamente si vanno a riordinare i blocchi (in cui è stato suddiviso il programma) ed infine si vanno ad inserire dei Corrector Slots, in modo da rendere stealthy il check sulla funzione di hash. Per capire questa tecnica, vediamo il seguente esempio:

Tamperproof (P,n):

1. Insert n checks: `if (hash(start,end)) RESPOND()` randomly in P
2. Randomize the basic blocks
3. Insert m ≥ n corrector slots: c1,...,cm
4. Compute n overlapping regions $1_1,...,l_n$: l_i = [$start_i$; end_i] and associate l_i with c_i
5. Associate checks with regions and set c_i such that hash(l_i)=0 (and a fingerpint?)



In questo modo, otteniamo che il blocco C è controllato da due funzioni di hash. Il fatto che tale blocco viene controllato due volte comporta un cosiddetto Overlap factor di 2 → ovvero vi è più di una funzione di hash che controlla un singolo blocco di codice e chiaramente maggiore è il numero di Overlap factor, maggiore è la sicurezza del codice.

Sorge a questo punto spontanea la domanda: “**Se proteggiamo il programma con questa famiglia di tampering che lavora per Introspection, come fa l’attaccante ha raggirarla?**” Di fatto, l’attaccante va a fare pattern-matching sulla struttura vista nell’esempio sopra (Figura 5), ovverosia l’attaccante va a fare pattern-matching su:

- start_address;
- end_address;
- loop che scorre tutta la porzione di codice di cui vogliamo calcolare l’hash;
- controllo if sul valore dell’hash e il valore aspettato.

Per l’attaccante, quindi, potrebbe essere semplice individuare dove effettivamente viene fatto il controllo di integrità del codice e chiaramente una volta individuato dove viene effettuato il controllo di integrità, per l’attaccante è facile bypassarlo.

Altri modi con cui l’attaccante può raggirare gli algoritmi di introspezione sono:

- andare a manomettere il calcolo dell’hash;
- andare a manomettere il check (dove per check, intendiamo il controllo tra il valore della funzione di hash e il valore atteso).

Quindi, l’attaccante per raggirare gli algoritmi di introspezione dovrebbe:

- analizzare il codice, al fine di individuare tutti i checkers;
- oppure analizzare il codice, al fine di individuare tutti i responders;

- rimuovere o disabilitare tutti i checkers o i responders, senza distruggere il resto del programma.

Per difendersi da questo possiamo:

- **costruire una rete di checkers e di responders**, ovverosia è bene non avere un solo checker e una sola responde, in modo tale da rendere anche più difficile per l'attaccante individuare tutti i checkers e i responders;
- **nascondere i valori di hash**, andando ad esempio utilizzare i Corrector Slots, in modo tale da rendere meno evidente il controllo di uguaglianza tra il valore di hash e quello atteso;
- **andare a correggere il codice**, nel caso in cui esso esso venga riconosciuto come manomesso.

Fino ad ora, però, ci siamo immaginati che gli attaccanti abbiamo solamente accesso al codice. In realtà, se ci immaginiamo **un attaccante che può modificare anche l'ambiente di esecuzione del codice, questo (ovvero l'attaccante) potrebbe raggirare qualsiasi algoritmo di introspezione**, perchè: quando noi calcoliamo l'hash, andiamo a leggere le istruzioni presenti in memoria e le trattiamo come dei dati (da notare la differenza, che quando prendiamo le istruzioni e le andiamo solamente a codificare, allora le istruzioni di fatto le trattiamo come dei dati. Quando, invece, andiamo ad eseguire le istruzioni, allora in questo caso le stiamo trattando di fatto come delle istruzioni). L'attaccante, quindi, fa una copia del codice e quindi si tiene:

- una copia del codice originale;
- una copia del codice manomesso.

e di fatto, abbiamo che:

- quando accediamo all'istruzione per calcolare l'hash, accediamo al codice originale;
- mentre quando accediamo all'istruzione per eseguire l'hash, accediamo al codice manomesso e quindi eseguiamo l'istruzione manomessa.

L'hash, quindi, viene calcolata nel modo corretto, ma effettivamente eseguiamo l'hash manomessa, in quanto l'attaccante ridireziona:

- gli **accessi in lettura** alle istruzioni della **copia originale**;
- gli **accessi in esecuzione** alle istruzioni della **copia manomessa**.

→ questo, chiaramente è un attacco all'ambiente in cui viene eseguito il codice (e non un attacco al codice).

Riassumendo, quindi, abbiamo che l'attaccante:

- copia il programma P in *Porig*;
- modifica P come desidera e inserisce la copia manomessa in *P'*;
- modifica il kernel del sistema operativo, in modo tale che la lettura dei dati avvenga su *Porig* e la lettura delle istruzioni su *P'*.

State Inspection

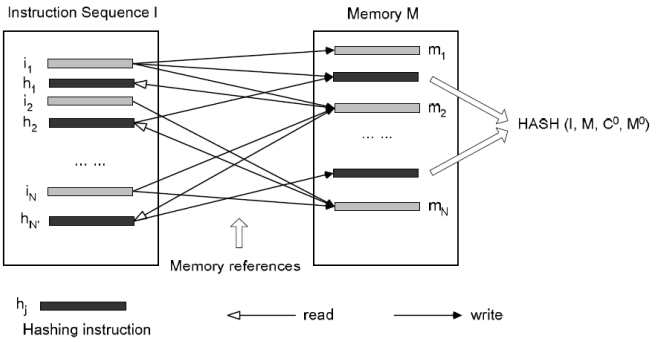
Un'altra famiglia di approcci al tampering è la **State Inspection** → dato che abbiamo visto, che l'introspezione (ovvero l'Introspection) legge il proprio codice, chiedendosi se il codice è corretto e di conseguenza:

- l'unico controllo è sul codice (in particolar modo sulla sintassi del codice);
- facile da raggirare per un attaccante, in quanto non è stealthy.

Invece, quindi, di chiedersi se il codice è corretto, **possiamo chiederci se il programma si trova in uno stato corretto** e questa è l'idea di base della State Inspection → ci stacchiamo, quindi, dal codice e ci concentriamo maggiormente sullo stato del processo e quindi guardiamo maggiormente se il valore calcolato è quello che ci aspettiamo che venga calcolato o meno. Delle possibili soluzioni per effettuare lo State Inspection sono:

- aggiungere dei controlli di asserzione → questa non è una soluzione facile, in quanto lo stato attuale del programma dipende da tutti gli stati precedenti e quindi è difficile trovare delle invarianti non banale da aggiungere e verificare;
- chiamare delle funzioni sensate sui challenge data (come per esempio la funzione di hash) → generare automaticamente, però, dei challenge data che esercitino aspetti importanti di una funzione non è facile e oltretutto, aggiungere questi dati nel programma in modo furtivo non è facile;
- **Oblivious Hashing** → l'idea alla base di questa soluzione è la seguente: invece di calcolare l'hash sul codice, viene calcolato l'hash sulla traccia di esecuzione e quindi, intuitivamente, possiamo capire che si tratta di una tecnica che ha più a che fare con quello che viene calcolato, rispetto a come effettivamente viene scritto il codice. Di conseguenza l'idea è la seguente: Se il programma originale, lo possiamo vedere come una sequenza di istruzioni che hanno accesso alla memoria e che vanno a leggere e scrivere sulla memoria, allora l'Oblivious

Hashing va ad arricchire ogni istruzione con il calcolo dell'hash di quella traccia di esecuzione.



Vediamo, allora, che inizia l'esecuzione, la quale è costituita da una sequenza di istruzioni eseguite una di seguito all'altra (vi saranno quindi, ad esempio, una serie di assegnamenti, poi una guardia, poi una chiamata a funzione, etc…). **Per ogni istruzioni da eseguire, oltre ad effettivamente eseguire l'istruzione in sè** (quindi andare ad assegnare un valore nel caso dell'assegnamento, oppure andare a controllare due valori nel caso della guardia) **si va a contribuire al calcolo dell'hash della traccia**.

Con l'Oblivious Hashing, quindi, l'hash viene calcolato sulla traccia di esecuzione. Chiaramente, ad un certo punto dell'esecuzione, vi saranno dei controlli, i quali andranno a controllare se il valore della traccia corrisponde o meno al valore presunto. Il problema di questa soluzione, è che in un programma si hanno tante tracce di esecuzione diverse e il valore di ciascuna traccia non è "deterministico", nel senso che il valore della traccia dipende da:

- ramo vero o falso che si è seguito;
- ambiente di esecuzione;
- input.

→ di questa soluzione di Oblivious Hashing (proposta nel 2002) se ne è capito il senso, in quanto calcolare l'hash della traccia poteva aver senso, ma all'interno del codice si devono inserire dei punti in cui viene controllato effettivamente il valore di tale hash. Ma se il valore dell'hash non è determinabile staticamente (in quanto dipende dall'input, dall'ambiente di esecuzione e dai rami seguiti), non è possibile inserire tali controlli sull'hash → in realtà, quindi, quello che effettivamente si può controllare con questa tecnica sono solamente i pezzi di traccia che vengono sempre eseguiti, ovvero i pezzi che sono calcolabili staticamente e quindi deterministici.

Nel 2018 questa soluzione è stata ripresa da alcuni studiosi dell'Università di Monaco, i quali hanno evidenziato che solamente il 13% delle istruzioni del programma potevano essere protette con la tecnica di Oblivious Hashing, in quanto solamente il 13% delle istruzioni avevano un'esecuzione deterministica. "**La restante percentuale di istruzioni (non calcolabili staticamente) come possiamo proteggerle?**" Si può prendere il grafo delle dipendenze, il quale evidenzia sia le dipendenze di controllo sia le dipendenze di valore e una volta esaminato il grafo delle dipendenze, siamo in grado di determinare quali istruzioni dipendono da dati non deterministici e quali no. L'idea alla base di questa soluzione è, quindi, quella di etichettare ogni istruzione come:

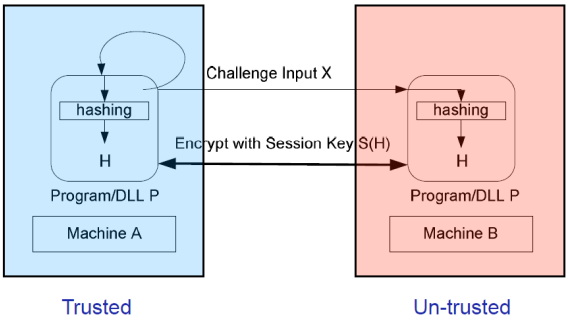
- input-independent;
- data-independent;
- data-dependent;
- control-flow dependent.

e realizzare l'Oblivious Hashing utilizzando variabili hash distinte per ogni sequenza ordinata di blocchi di base ordinati, che vengono eseguiti rigorosamente nell'ordine identificato, in ogni esecuzione. Cioè, per un dato blocco nella sequenza, tutti i blocchi precedenti sono necessariamente eseguiti prima di raggiungere il blocco.

Questa soluzione ha come **vantaggio** il fatto che è legata alla semantica e a ciò che viene calcolato e questo ha come conseguenza diretta, il fatto che è difficilmente raggirabile da parte di un attaccante. Lo **svantaggio** è, che per un programma vi sono molteplici tracce di esecuzione e per ognuna di queste vi è una diversa hash.

Remote Tamper-proofing

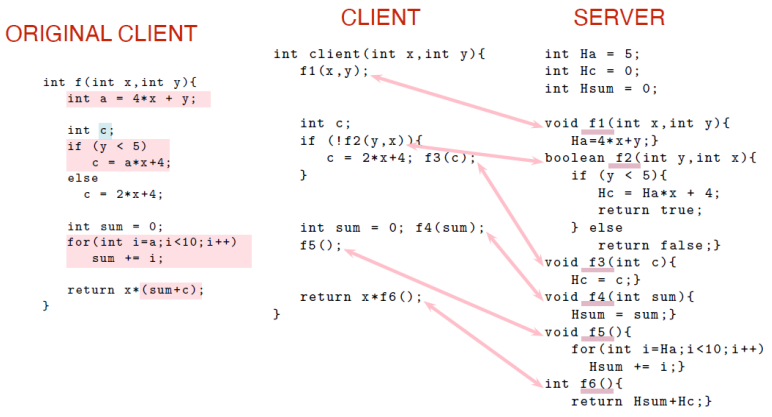
Tecniche che si immaginano di avere un Client e un Server che comunicano tra loro e quindi avere un'applicazione lato Client e un'applicazione lato Server. Ci mettiamo nello scenario, in cui il Server sia fidato e che, invece, il Client non sia fidato e di conseguenza ci chiediamo: "**Come può il Server fidarsi del fatto, che il Client stia eseguendo una versione non manomessa dell'applicazione?**" Abbiamo, quindi, che il Client è in costante comunicazione con il Server e il nostro obiettivo è sfruttare tale comunicazione, al fine di verificare l'integrità del codice in esecuzione sul Client (dato che ci ricordiamo, che nel nostro scenario, il Client non è affidabile e di conseguenza esso può mentire). Per riuscire a fare questo, si può utilizzare l'Oblivious Hashing e ricevere i valori di challenge da remoto per l'hashing. L'hashing, poi, può essere computato normalmente.



Come si può vedere dall'immagine sopra, l'idea alla base di questa soluzione, è quella di interrogare il Client, attraverso le challenge e le response, e controllare se risponde in maniera corretta. Il fatto, però, è che il Client può mentire, essendo che esso non è fidato e di conseguenza il Client può utilizzare una versione del codice originale in cui viene calcolata l'hash e una versione manomessa in cui viene l'hash viene eseguita → come soluzione a questo problema, era stato implementato un algoritmo, il quale voleva proteggere la proprietà intellettuale del codice. L'idea alla base di questo algoritmo partiva dal presupposto, che il Server era fidato e il Client era non fidato e consisteva nel suddividere il codice in:

- **open components** → installati ed eseguiti su macchine non affidabili e quindi nel nostro scenario sul Client;
- **hidden components** → installati ed eseguiti su macchine fidate e quindi nel nostro scenario sul Server.

Per l'attaccante, il quale ha accesso solamente al Client, è difficile ricostruire l'applicazione intera, senza conoscere le hidden components, le quali si trovano solamente sul Server.



Orthogonal Replacement

Abbiamo detto, che quando andiamo ad offuscare il codice, rendiamo più complicato per l'attaccante comprenderlo. Idealmente, dopo un determinato periodo di tempo, dovrei dare all'attaccante una nuova versione offuscata del codice, in quanto dopo un certo periodo di tempo, l'attaccante sarà in grado di comprendere la prima versione offuscata del codice → dovremmo, quindi, periodicamente andare a sostituire il programma con una nuova versione offuscata (un esempio di ciò, lo si poteva vedere con Skype, il quale qualche anno fa continuava a rilasciare aggiornamenti dell'applicazione, i quali in realtà andavano a modificare la versione del Client che veniva rilasciata). Sorge a questo punto spontanea la domanda: "**É vero che tutto quello che ho capito analizzando la prima versione dell'offuscamento del programma, non mi è di aiuto a capire la seconda versione dell'offuscamento?**" L'idea, quindi, alla base dell'Orthogonal Replacement, è quella di sostituire, con una **variante ortogonale**, una versione dell'offuscamento da quella successiva. Con variante ortogonale, si voleva catturare l'idea, che l'attaccante analizzando le vecchie versioni dell'offuscamento, non ha alcun vantaggio nel capire le nuove versioni dell'offuscamento. "**Cosa significa, però, aggiungere questa variante ortogonale?**" Un primo significato, potrebbe essere quello che:

- due frammenti di codice sono ortogonali, se gli algoritmi di clone-detection vengono ingannati, ovverosia se gli algoritmi di clone-detection non si accorgono che i due frammenti di codice sono uno il clone dell'altro.