Protezione del software: limiti e potenzialità

In *cybersecurity* siamo abituati a confinare le informazioni all'interno di perimetri, o **zone di sicurezza**. All'interno della zona di sicurezza, considerata un ambiente attendibile, è possibile accedere alle informazioni e utilizzarle in una forma condivisa comune, in modo sicuro e aperto. Al di fuori della zona, le informazioni non dovrebbero essere accessibili senza autorizzazione. Il problema che stiamo affrontando al giorno d'oggi è che, una volta che qualcuno penetra nella zona di sicurezza, nulla può fermare l'accesso e l'uso non autorizzati delle informazioni.

È possibile costruire una zona di sicurezza attorno al SW – ovvero tale per cui la logica e le strutture dati, cioè il know-how che ha reso possibile la sua realizzazione – **siano protette?**

Ci sono alcune caratteristiche intrinseche del Software (SW) che lo rendono un prodotto altamente vulnerabile sotto quest'aspetto. Innanzitutto il fatto che esso incorpora e rende per certi versi esplicito il know-how che ha reso possibile la sua creazione. Tutto è esplicitato in modo chiaro nelle istruzioni e nelle strutture dati che compongono un dato programma o una data architettura SW. Pertanto, quanto più il know-how di una azienda è concentrato sul prodotto SW, tanto più questo è a disposizione di chiunque ne faccia una copia, lecita o illecita che sia!
Inoltre, il SW per essere eseguito (o interpretato) deve essere esplicitato in un **linguaggio** (es. C, C++. .Net. Java, binario x86 etc). Pertanto, non possiamo immaginare di criptare le istruzioni e aspettarci che il programma così reso illeggibile, sia eseguibile da un interprete (sia esso in PC o altro) senza che questo, a un certo punto, non decripti le istruzioni rendendole in questo modo leggibili [1].

Un attacco **MATE** (*Man-At-The-End*) significa la possibilità, assolutamente concreta, che un attaccante abbia a disposizione un dato sistema SW, che possa analizzarlo – ad esempio in un *debugger* o mediante emulazione – possa disassemblarlo e decompilarlo, nel caso di SW distribuito come file eseguibili, estraendo la struttura logica che lo caratterizza, e possa fare *reverse engineering* delle strutture dati e quindi del know-how che ha respo possibile la sua realizzazione [4]. Attacchi MATE sono frequenti, se non la norma, sia nel mercato legale che nel mercato illegale.
Tipici attacchi MATE sono: individuazione ed eventuale riutilizzo di funzionalità ritenute strategiche, identificazione di falle o contesti di utilizzo che possono essere sfruttati per compromettere le funzionalità del SW, o pirateria e compromissione del'IP (*Intellectual Property*).

Se pensiamo al SW come a un *asset* fondamentale del nostro business, non possiamo prescindere da considerare nel suo sviluppo anche metodi e tecnologie di protezione. Sono necessarie tecnologie che combinino crittografia, trasformazione di programmi e ingegneria del software al fine di permettere una **sicurezza incentrata sull'applicazione**, dove la sicurezza di un'applicazione non è supportata solo dalle funzionalità di sicurezza provenienti dall'ambiente in cui l'applicazione è in esecuzione, ma è anche integrata nel suo stesso codice, utilizzando tecnologie di protezione appositamente progettate per il software.

Un risultato teorico fondamentale – apparso in versione preliminare nel 2001 e in versione estesa e dettagliata nel 2012 [2] (si veda [1] per una introduzione) – dimostra che è teoricamente impossibile trasformare un dato programma P mediante un compilatore T in modo che il programma compilato T(P) mantenga sostanzialmente le performance di P e sia tale che tutto ciò che un attacco MATE a T(P) può imparare è solo la relazione input/output di T(P), detto *virtual black-box* (VBB). Più semplicemente, questo risultato dimostra che non è

possibile costruire un compilatore (che chiameremo offuscatore) che renda il nostro programma (*asset*) P completamente oscuro, ovvero per il quale tutta la logica, la dinamica delle strutture dati sia nascosta e sia visibile solo ciò che P calcola.

Nella scienza è ben noto che i risultati negativi sono spesso fonte dei più importanti sviluppi, anche tecnologici. L'impossibilità di avere un offuscatore VBB chiaramente pone dei limiti alla possibilità di proteggere il nostro SW. Ad esempio non possiamo immaginare di nascondere in modo indefinito un segreto (ad esempio una chiave criptografica) in un programma e assumere che questa non sia determinabile da un attacco MATE.
Bisogna quindi **pensare alla protezione del SW non come un processo definitivo**, ovvero dato in modo definitivo da un magico compilatore offuscante VBB T per un qualsiasi attacco MATE, come è nel caso della criptografia per ciò che concerne la comunicazione sicura tra parti rispetto ad attacchi *Man-In-The-Middle*. Dobbiamo pensare in modo differente, sfruttando le intrinseche proprietà del SW, che in questo caso ci possono venire in aiuto:

1. i programmi sono probabilmente tra gli oggetti più complessi mai costruiti dall'uomo. Anche comprendere un programma di poche righe può essere estremamente difficile;
2. per comprendere un programma abbiamo bisogno di strumenti che, a causa del punto 1), sono anch'essi programmi: disassemblatori per ricostruire il codice simbolico da un dato eseguibile, decompilatori per ricostruire il sorgente da un codice simbolico eseguibile [2], analizzatori statici e dinamici, *debugger*, emulatori, *slicer* per ridurre la struttura etc.

In quest'ottica, proteggere il SW significa sviluppare metodi e tecnologie per **rendere il SW oltremodo complicato** e per far

sì che gli strumenti dell'attaccante siano inefficaci a ridurre la complessità del problema del *reverse engineering*, rendendo questo processo estremamente dispendioso e più lungo nel tempo. Quindi significa pensare la protezione in un arco temporale: prima o poi l'attaccante MATE avrà la meglio.

Questo è il campo di studio e ricerca del *code obfuscation*, ovvero dei metodi e delle tecnologie per rendere parzialmente possibile il nascondimento di informazioni nel programma. Si tratta spesso di una combinazione di tecniche che uniscono al mero nascondimento alcune funzionalità aggiuntive. Ad esempio, offuscamento e versionamento frequente possono essere strategie vincenti per ingaggiare l'attaccante in una rincorsa continua a tutto svantaggio di chi insegue, in questo caso l'attaccante!

Sul piano tecnologico, le tecnologie impiegate sono principalmente una combinazione di trasformazioni di programmi, ovvero manipolazioni che mantengono sostanzialmente inalterato le funzionalità dei programmi trasformati, agendo quindi sulla struttura del codice. Un'antologia di queste tecniche si può trovare in [3].

L'introduzione di funzionalità **anti manipolazione** (*anti-tampering*) che permettano al codice di ispezionare e verificare a *run-time* la propria consistenza contrastando in questo modo sia riutilizzazioni non autorizzate che attacchi MATE svolti mediante *debugger*; l'iniezione di SW *watermark*, ovvero di particolari sequenze che identificano in modo univoco un dato programma e che pertanto possono essere utilizzate per tracciarne l'eventuale riutilizzo, sono esempi di tecnologie che richiedono offuscamento. Offuscamento di parti del programma, dal layout (offuscamento banale riguardante nomi ad esempio di variabili e strutture dati) passando per la logica di esecuzione (offuscamento più complesso spesso ottenibile mediante un processo di virtualizzazione controllata su architetture costruite *ad hoc* per essere complesse e non standard) fino alle strutture dati

(offuscamento dei dati mediante trasformazioni omomorfe).

Queste tecnologie devono essere pensate e adattate al particolare contesto e spesso progettate *ad hoc*, onde evitare che per ogni tecnologia di protezione disponibile sul mercato esista una tecnologia simmetrica che ne vanifichi le funzionalità.

Note

[1] Su questo aspetto studi recenti cercano di estendere tecniche di *homomorphic encryption* [1], ovvero la possibilità di operare su dati criptati senza una fase di decriptazione, alla esecuzione di programmi. Risultati preliminari rendono possibile idealmente l'esecuzione di istruzioni criptate su macchine a stati finiti di potenza limitata. L'estendibilità di questa tecnologia a set di istruzioni più ampie, o a veri e propri linguaggi di programmazione, è oggetto di studio e comunque fuori dalla portata dell'odierna ingegneria del SW.

[2] Chiunque pensasse che distribuire l'eseguibile sia una forma di protezione del SW è chiaramente ingenuo. Ogni eseguibile se non opportunamente manipolato (offuscato) è decompilabile.

Riferimenti

[1] B. Barak. Hopes, fears, and software obfuscation. Communications of the ACM, 2016 vol. 59(3), pp. 88-96.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S.P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6, 2012.

[3] C. Collberg. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley, 2010.

[4] C. Collberg, J. Davidson, R. Giacobazzi, Y. Gu, A.

Herzberg, and F. Wang. **Towards Digital Asset Protection**. IEEE Intelligent Systems. 26(6):8-13, 2011.

Articolo a cura di **Roberto Giacobazzi**

https://www.ictsecuritymagazine.com/articoli/protezione-del-software-limiti-e-potenzialita/