Sicurezza Informatica

Progetto d'esame

Lorenzo Nodari - Mat. 715263

A.A. 2021/2022

Background

Introduzione a Mimikatz

Mimikatz nasce nel 2007 ad opera di Benjamin Delpy come semplice progetto per sperimentare con la sicurezza di Windows. Da allora, è diventato uno dei più noti e utilizzati software per la post-exploitation in ambienti Windows grazie alle numerose funzionalità offerte. Esso include infatti diversi moduli per:

- estrazione di vari tipi di credenziali;
- privilege escalation locale;
- attacchi a domini Active Directory (DcSync, DcShadow);
- interazione con sistemi di storage cifrato (DPAPI);
- attacchi pass-the-hash, pass-the-cache e pass-the-ticket;
- attacchi al protocollo di autenticazione Kerberos (GoldenTicket, Skeleton-Key);
- *molto* altro.

Dato la sua elevata versatilità e la sua natura open-source, Mimikatz è stato e viene tutt'ora utilizzato come componente all'interno di diversi malware[1], al fine di arricchirne le capacità offensive. Per tale motivo, la capacità di rilevare l'esecuzione di Mimikatz risulta uno stumento fondamentale nell'arsenale difensivo di qualsiasi organizzazione.

Un approccio basato sulle syscall

Alla luce dalle considerazioni esposte nella sezione precedente, lo scopo del mio progetto d'esame è stata la sperimentazione di una strategia di detection basata sull'analisi delle chiamate di sistema – *syscalls* – effettuate da Mimikatz durante l'esecuzione. Tale scelta è stata motivata, in primo luogo, da un'analisi della

[1] Ad esempio, si veda `https://attack.mitre.org/software/S0002/` per un elenco di APTs note che han fatto ricorso a Mimikatz.

letteratura correntemente disponibile[2] sul tema della detection di Mimikatz, durante la quale ho osservato una carenza di approcci simili.

La maggior parte delle fonti da me identificate, infatti, presenta degli *indicators of compromise* (IoCs) ottenuti da valutazioni empiriche legate a specifici aspetti del funzionamento di Mimikatz. Ad esempio, al fine di rilevare l'esecuzione del modulo **sekurlsa::logonpasswords**, adibito alla lettura delle credenziali contenute all'interno della Local Security Authority di Windows, la strategia comunemente consigliata consiste nel monitorare i tentativi di accesso della memoria del relativo processo: lsass.exe. I principali vantaggi di tale approccio sono:

- **semplicità**: il monitoraggio di tale attività può essere facilmente implementato mediante l'utilizzo di Sysmon[3] e integrato con sistemi SIEM per la correlazione degli eventi;
- **affidabilità**: poichè l'accesso al processo risulta fondamentale per il funzionamento del modulo, la corretta esecuzione dello stesso non può avvenire senza che tale evento venga generato;
- **specificità**: tale attività è tendenzialmente rara da osservare, durante la normale operatività dei sistemi, da parte di processi non malevoli e ciò aiuta a garantire un basso numero di falsi positivi;

Tuttavia, oltre ai vantaggi descritti, possiamo identificare alcuni aspetti problematici di tale approccio.

In primo luogo, esso risulta applicabile alle sole funzionalità di Mimikatz caratterizzate da interazioni "notevoli" con l'ambiente: accesso ad altri processi, modifiche al registro di sistema, caricamento di DLL, generazione di traffico di rete, etc... Sebbene tale prerequisito sia rispettato da alcuni dei moduli di Mimikatz più pericolosi, altri[4] operano rimanendo interamente confinati nella memoria del processo, costringendo cosí i *Blue Teams* a ricorrere a strategie di detection basate su IoCs più generici e quindi meno affidabili.

In secondo luogo, a causa della necessità di rilevare e correlare artefatti e/o eventi generati dall'esecuzione di Mimikatz, tale strategia risulta tendenzialmente reattiva: il rilevamento dell'attività malevola è infatti spesso possibile solo successivamente all'avvenuta esecuzione, quando ormai i potenziali danni legati all'attacco sono stati fatti.

Sulla base di queste considerazioni ho pertanto deciso di sperimentare un approccio di detection basato sul tracciamento delle chiamate di sistema effettuate dal processo di Mimikatz. Quest'ultime infatti, in quanto interfaccia tra il sistema operativo e i programmi user-mode, rappresentano un elemento necessario e imprescindibile affinchè Mimikatz possa effettuare qualsiasi operazione di suo interesse, e, più in generale, possono essere utilizzate come "firma" delle operazioni effettuate da un generico malware, come testimoniato da [9].

[2] Si faccia riferimento alla bibliografia per l'elenco completo delle fonti da me consultate.
[3] Si veda: `https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon`.
[4] Si pensi, ad esempio, ai moduli adibiti alla generazione di GoldenTicket per Kerberos.

Un approccio simile, sebbene più complesso a livello implementativo, potrebbe quindi permettere di superare le limitazioni esposte, garantendo:

- la possibilità di generare una "firma" per moduli arbitrari di Mimikatz, prescindendo dagli specifici artefatti ed eventi di sistema generati dall'esecuzione del software;
- la possibilità di effettuare la detection in modo proattivo: mediante l'implementazione di una sandbox, un eventuale processo malevolo potrebbe essere terminato non appena venisse rilevata la presenza della firma precedentemente identificata, garantendo quindi un processo di *incident response* più tempestivo e una migliore mitigazione, o addirittura prevenzione, dei possibili danni;

senza sacrificare la specificità e l'affidabilità del processo di detection.

Nel resto di questo documento presenterò pertanto la strategia di generazione e rilevazione delle firme di esecuzione da me sviluppata e i risultati ottenuti sperimentalmente dall'applicazione di tale approccio a tre diversi moduli di Mimikatz.

Metodologia

Questo capitolo documenta le principali decisioni progettuali prese durante lo svolgimento del lavoro.

Moduli analizzati

La scelta dei moduli da analizzare è stata guidata dalla volontà di valutare l'applicabilità del metodo proposto a tre situazioni caratterizzate da un diverso grado di qualità degli IoCs già noti in letteratura. In particolare, ho voluto identificare:

1. un modulo per il quale sono già noti IoCs di buona qualità;
2. un modulo per cui gli IoCs noti presentano qualche aspetto problematico;
3. un modulo per il quale non sono disponibili in letteratura IoCs specifici;

sekurlsa::logonpasswords

Nell'ottica appena esposta, il modulo scelto per il primo caso è stato **sekurlsa::logonpasswords** – uno tra i più noti e utilizzati di Mimikatz – responsabile, come già anticipato, della funzionalità di estrazione di vari tipi di credenziali dalla memoria della Local Security Authority di Windows: dalle password clear-text degli account utente locali fino ai ticket Kerberos utilizzati per l'autenticazione presso i servizi all'interno dei domini Active Directory.

A causa alla notorietà di tale modulo, la letteratura propone varie strategie di detection caratterizzate da diversi livelli di raffinatezza. Tra queste, quelle che ho trovato di maggior valore sono:

- il monitoraggio delle richieste di accesso al processo lsass.exe con permessi QUERY_LIMITED_INFORMATION e VM_READ;
- il monitoraggio del caricamento da parte dei processi di una lista di DLL caratteristica di Mimikatz;

Oltre alla disponibilità degli IoCs esposti, ulteriori fattori che mi hanno portato alla scelta di questo modulo sono stati la sua semplicità di utilizzo, l'alto impatto ad esso associato e la sua elevata diffusione.

lsadump::dcsync

Come rappresentante per il secondo caso ho scelto di analizzare il modulo **lsadump::dcsync**, implementazione dell'omonimo attacco, grazie al quale Mimikatz è in grado di impersonificare un Domain Controller e richiedere ad un altro Domain Controller la replicazione di tutti i dati contenuti nel dominio, tra cui gli hash delle password degli account. Tra questi, risulta particolarmente critico l'hash della password dell'account **krbtgt** – Kerberos Ticket Granting Ticket – in quanto la sua conoscenza permette all'attaccante di generare *Golden Tickets*, token di autenticazione utilizzabili per effettuare sostanzialmente qualsiasi operazione all'interno del dominio.

Le principali strategie per la detection di tale modulo sono due:

- monitoraggio, a livello di rete, del traffico RPC[5] finalizzato all'invocazione della funzione "DRSGetNCChanges", appartenente alle API del protocollo di replicazione dei dati utilizzato all'interno dei domini Active Directory;
- auditing dei tentativi di accesso ai directory services forniti dai Domain Controller all'interno del dominio.

La prima strategia, sebbene più affidabile e pertanto preferibile, è tendenzialmente più complessa da implementare a causa della necessità di utilizzare un sistema IDS/IPS e una specifica segmentazione logica della rete. Al contrario, la seconda strategia è estremamente semplice da adottare poichè richiede unicamente l'abilitazione, mediante Group Policy, della registrazione dell'Event ID 4662 all'intero dell'Event Logger integrato di Windows. Tuttavia, a causa della natura estremamente generica di tali eventi – generati per ogni operazione effettuata su un *qualsiasi* oggetto all'interno del dominio – questa strategia risulta meno efficace.

Nonostante ciò, a causa degli strumenti a mia disposizione nell'ambiente di test utilizzato per la fase sperimentale, ho scelto di prendere come riferimento gli IoC legati alla seconda strategia. Fortunatamente, ritengo che tale scelta non abbia leso la validità degli esperimenti effettuati in quanto il principale limite per la detection di attacchi DcSync è comune ad entrambi gli approcci: poichè l'attività di replicazione dei dati di dominio è una componente integrante della strategia di fault-tolerance adottata dalle tecnologie Active Directory, il traffico ad esso associata è osservabile durante la normale operatività dei sistemi e comporta pertanto la presenza di una componente di rumore significativa durante l'analisi degli eventi potenzialmente associati ad attività malevola.

token::elevate

Infine, come terzo modulo, ho scelto di analizzare **token::elevate**, mediante il quale è possibile elevare i permessi associati al processo di Mimikatz tramite l'impersonificazione del token di sicurezza di un altro processo. In particolare, tramite tale modulo, Mimikatz è in grado di:

[5] Remote Procedure Call

- impersonificare l'utente NT Authority\SYSTEM, al quale è associato il massimo grado di controllo sul computer locale;
- cercare automaticamente e – se trovato – impersonificare il token associato a un account che gode di privilegi amministrativi all'interno del dominio;

ottenendo cosí i permessi necessari a compiere la quasi totalità degli attacchi disponibili. Per questo motivo, possiamo considerare tale modulo "ausiliario", nel senso che il suo utilizzo è sempre strumentale all'esecuzione di altri moduli.

Come anticipato, la principale motivazione dietro alla scelta di questo modulo è l'assenza di IoC ad esso specifici nella letteratura, probabilmente causata – almeno in parte – proprio dalla sua natura ausiliaria.

In ogni caso, prescindendo dall'effettiva motivazione legata a tale vuoto nella letteratura, ritengo quindi che questo modulo possa rappresentare un ottimo candidato per lo studio della validità del mio approccio di detection applicato a un caso in cui l'approccio "tradizionale" non è utilizzato o utilizzabile.

Fase sperimentale

Una volta stabiliti i tre moduli da analizzare, ho provveduto a organizzare lo svolgimento dei vari esperimenti, definendo quanto riportato di seguito.

Ambiente di test e toolset

Al fine di svolgere i vari esperimenti in modo controllato ho predisposto un ambiente di test composto da un semplice dominio Active Directory comprendente due macchine virtuali:

- la prima, dotata di sistema operativo Windows 10 Pro[6] e utilizzata per l'esecuzione dei vari moduli di Mimikatz;
- la seconda, dotata di sistema operativo Windows Server 2019 Standard[7] e utilizzata come Domain Controller del dominio;

Per quanto riguarda invece il toolset utilizzato, esso è stato composto da:

- **drstrace**: tool open-source per il tracciamento delle chiamate di sistema effettuate da un processo durante l'esecuzione, fonte primaria dei dati utilizzati per la generazione delle firme di esecuzione di ciascun modulo analizzato;
- **Sysmon**: tool per il monitoraggio di vari eventi – creazione di processi, caricamento di DLL, accesso a processi esistenti, etc... – non registrati dai normali log di Windows. Grazie ad esso ho potuto raccogliere i dati che mi hanno permesso di effettuare un'analisi degli IoC noti per i primi due moduli analizzati;

[6] Versione 21H2, Build 19044.2364
[7] Build 17763.rs5_release.180914-1434

- **Windows Event Viewer**: tool predefinito di Windows per la visualizzazione dei log di sistema, grazie al quale ho potuto filtrare ed esportare i log generati da Sysmon per l'analisi;
- **Python 3.9**: linguaggio di scripting grazie al quale ho automatizzato la maggior parte dei task che ho dovuto svolgere, a partire dalla raccolta dei dati fino alla generazione e verifica delle firme per la detection;

A questi va ovviamente aggiunto Mimikatz, utilizzato nella versione 2.1.1.

Analisi preliminare

Al fine di verificare l'effettiva bontà – o problematicità – degli IoC proposti dalla letteratura per i primi due moduli di Mimikatz analizzati, ho condotto una prima fase di analisi durante la quale ho tentato di valutarne l'affidabiltà e la specificità, quantificate rispettivamente mediante:

- numero di esecuzioni per le quali l'IoC sotto esame è stato correttamente rilevato;
- numero di eventi – appartenenti a categorie monitorate – registrati nei log ma *non* generate da un'esecuzione di Mimikatz. Con tale quantità ho voluto effettuare una stima del rumore caratteristico di ciascuna strategia di detection.

Gli IoC sotto esame sono stati raccolti mediante l'utilizzo di Sysmon[8] e analizzati tramite il codice contenuto nel file **sysmon_traces.py**, in particolare tramite la funzione **analyze_sysmon_traces()**, definita dalle righe 85-158.

Struttura degli esperimenti

Terminata l'analisi preliminare degli IoC noti ho condotto l'effettiva sperimentazione della strategia di detection proposta, suddividendo gli esperimenti associati a ciascun modulo in diverse *sessioni* – riassunte nella tabella 1 – al fine di testare diverse condizioni sperimentali per ognuna di esse. Per una descrizione esaustiva delle esatte condizioni sperimentali associate a ciascuna sessione, si faccia riferimento al file **experiments.txt**.

Modulo	# Sessioni	# Esecuzioni per sessione	Var. Indipendente
sekurlsa::logonpasswords	6	100	Numero e tipologia di utenti loggati sulla macchina + Presenza di credenziali clear-text nella memoria di lsass.exe
lsadump::dcsync	2	150	Dati estratti dal Domain Controller
token::elevate	2	50	Account impersonificato

Tabella 1: Tabella riassuntiva delle diverse sessioni atomiche di sperimentazione condotte per ciascun modulo di Mimikatz analizzato.

[8]Si veda il contenuto della cartella **sysmon_configs** per gli specifici file di configurazione utilizzati

La ripetuta esecuzione di ciascun modulo e la raccolta dei file di log contenenti le syscall effettuate dal processo di Mimikatz per ciascuna di esse sono state effettuate in modo automatico mediante l'ausilio dello script **data_collection.py**. I dati cosí ottenuti sono quindi stati analizzati al fine di produrre le firme utilizzate per la detection mediante il codice contenuto nel file **syscall_traces.py**, approfondito nella sezione successiva.

Infine, i dati ottenuti da più sessioni sono stati aggregati e ri-analizzati al fine di produrre nuove firme, con lo scopo di generalizzare i risultati ottenuti per ciascun modulo e rendere queste ultime meno dipendenti dalle specifiche condizioni sperimentali testate. La diverse sessioni aggregate cosí create sono riassunte dalla tabella 2. La funzionalità di aggregazione dei risultati è implementata dalla funzione **aggregate_syscall_traces_analysis()**, definita dalle righe 399-418 del file **syscall_traces.py**. Analogamente è stato quindi fatto per gli IoC ottenuti tramite Sysmon mediante la funzione **aggregate_sysmon_traces_analysis()** definita dalle righe 161-201 del file **sysmon_traces.py**.

Sessione	Descrizione	# Esecuzioni aggregate	ID Sessioni aggregate
sekurlsa::logonpasswords::cleartext	Firma ottenuta con storage delle credenziali clear-text abilitato	300	1 + 5 + 6
sekurlsa::logonpasswords::no_cleartext	Firma ottenuta con storage delle credenziali clear-text disabilitato	300	2 + 3 + 4
sekurlsa::logonpasswords::all	Firma generica per il modulo sekurlsa::logonpasswords	600	1 + 2 + 3 + 4 + 5 + 6
lsadump::dcsync::all	Firma generica per il modulo lsadump::dcsync	300	1 + 2
token::elevate::all	Firma generica per il modulo token::elevate	100	system + domainadmin

Tabella 2: Tabella riassuntiva delle diverse sessioni aggregate generate per ciascun modulo di Mimikatz analizzato.

Infine, ho effettuato un benchmark delle firme cosí ottenute, applicando la strategia di detection ai file di log generati mediante l'esecuzione – tramite **drstrace** – di più software, al fine di testarne le prestazioni in diversi scenari. In particolare, il benchmark è formato dal tracciamento di:

- **Firefox, Chrome ed Edge**, ciascuno aperto e utilizzato per navigare a www.google.it e utilizzati per valutare il comportamento delle firme se applicate a software comune;
- **OpenOffice Calc e Writer**, scelti come esempi di normale software d'ufficio;
- **WinDBG e x64dbg**, entrambi aperti e utilizzati per effettuare l'attach al processo lsass.exe al fine di valutare il caso di sofware benigno ma caratterizzato da un comportamento sospetto;
- **Mimkatz #1**: esecuzione del modulo **sekurlsa::logonpasswords** finalizzata a testare la rispettiva firma;
- **Mimkatz #2**: esecuzione parziale del modulo **sekurlsa::logonpasswords**, seguita dal comando **token::whoami** e quindi ripresa e terminata. Questa istanza è utilizzata per dimostrare l'utilità della strategia di detection **lax**;

- **Mimkatz #3**: esecuzione del modulo **lsadump::dcsync** finalizzata a testare la rispettiva firma;
- **Mimkatz #4**, esecuzione del modulo **token::elevate** finalizzata a testare la rispettiva firma;
- **Mimkatz #5**, esecuzione del modulo **event::clear** finalizzata a testare la capacità delle firme generate di generalizzare la detection a moduli mai osservati prima.

Strategia di detection

Una volta raccolti, i file di output generati da **drstrace** per ciascuna esecuzione di Mimikatz sono analizzati al fine di generare le firme utilizzate per la detection dei vari moduli, seguendo i passi descritti di seguito. In particolare, il codice di tutte le funzioni qui descritte è disponibile all'interno del file **syscall_traces.py**.

Pre-processing dei dati

Come prima cosa, la funzione **get_syscall_sequences()** – definita alle righe 167-191 – è utilizzata per effettuare il parsing dei dati contenuti all'interno di ciascun file. Nonostante l'output di **drstrace** contenga, per *ciascuna* syscall effettuata:

1. il nome;
2. il valore di ciascun argomento;
3. l'esito: successo o fallimento;

ho deciso di utilizzare unicamente il primo tra questi dati per la generazione delle firme. Tale scelta è stata motivata dalla volontà di mantenere la strategia di detection il più semplice possibile e valutare un aumento di complessità unicamente nel caso in cui questa si fosse rivelata di qualità insoddisfacente.

Oltre al filtraggio delle informazioni sopracitate, tale funzione si occupa della compressione di gruppi di chiamate successive alla medesima syscall in un'unica istanza della stessa, esemplificata dalla figura 1. L'assunzione implicita dietro a questo accorgimento – giustificato euristicamente e finalizzato a semplificare ulteriormente le sequenze analizzate – è che chiamate ripetute alla medesima syscall siano probabilmente orientate, ad alto livello, al raggiungimento del medesimo scopo.

Generazione delle firme

Le sequenze di syscall cosí ottenute sono quindi state utilizzate come input effettivo del processo di analisi e generazione delle firme, implementato dalla funzione **analyze_sequences()**, definita dalle righe 317-396.

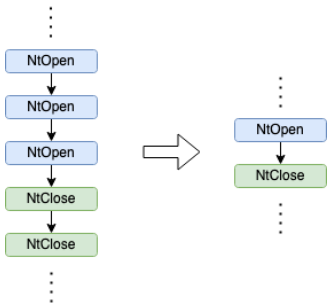

Figura 1: Esempio del processo di compressione delle sequenze di syscall.

Come prima cosa, viene effettuato un plot del numero di sequenze, sia totali che uniche[9], rilevate per ciascuna lunghezza osservata. Grazie a tale plot è possibile ottenere una stima qualitativa dell'esaustività dei dati raccolti: qualora infatti non fossero presenti sequenze duplicate sarebbe ragionevole valutare se il numero di esecuzioni effettuate possa non essere stato sufficiente a garantire l'osservazione di tutti i percorsi d'esecuzione possibili per il software analizzato[10].

Successivamente viene quindi identificato l'insieme di tutte le syscall invocate e, per ciascuna di esse, vengono calcolati il numero di invocazioni osservate in *tutte* le sequenze e, conseguentemente, la *media* e la *deviazione standard* di tale quantità. A questo punto, le sequenze sono filtrate, mantenendo in ciascuna di esse solo le syscall eseguite lo stesso numero di volte in *tutte* le esecuzioni osservate e caratterizzate pertanto da deviazione standard pari a zero, definite *syscall fisse*.

Infine viene determinata la sequenza più comune tra quelle rimanenti, che rappresenterà quindi la *firma di esecuzione* effettivamente utilizzata per rilevare il modulo analizzato.

Detection delle firme

Una volta generate, le firme sono utilizzabili per la detection mediante la funzione **check_signature_in_tracefile()**, definita dalle righe 97-164. Come prima cosa, la sequenza di syscall contenuta nel file di output di **drstrace** associato al processo sospetto è compressa e filtrata sulle sole syscall *fisse*, esattamente come fatto durante l'analisi. A questo punto, la sequenza risultante può essere confrontata con la firma considerata, per ciascun offset tra le due, secondo due possibili strategie:

- **strict**: ciascun confronto è interrotto al primo indice per cui le due sequenze differiscono;

[9] *i.e:* non considerando eventuali sequenze duplicate
[10] condizione che porterebbe necessariamente alla registrazione di sequenze duplicate.

- **lax**: quando viene trovata una posizione per cui la sequenza analizzata e la firma differiscono, la prima viene fatta scorrere in avanti e il confronto prosegue;

e la probabilità della presenza della firma all'interno del campione analizzato è stimata mediante la massima lunghezza di un match ottenuto al passo precedente.

Definendo infine un limite inferiore alla confidenza richiesta per la positività è quindi possibile ottenere una risposta in termini di avvenuta o mancata detection.

Risultati

Analisi preliminare

Per quanto riguarda il modulo **sekurlsa::logonpasswords**, ho condotto l'analisi preliminare considerando congiuntamente entrambi gli IoC precedentemente esposti. In particolare, la lista delle DLL monitorate è stata derivata da [5]. I risultati ottenuti sono osservabili in figura 2: sulla sinistra abbiamo il numero di detection per ciascun singolo IoC, mentre sulla destra abbiamo la quantità di rumore rilevata all'interno dei log ottenuti tramite Sysmon, sia totale che aggiustata sulla base del rumore introdotto dalla procedura di raccolta dei dati stessi.

Possiamo quindi notare come tali IoC siano effettivamente estremamente affidabili – al 100% secondo i dati a disposizione – ad eccezione di alcune DLL, evidentemente non necessarie per l'esecuzione del modulo in questione e quindi non caricate dal processo. Per quanto riguarda invece il rumore, notiamo come questi si attesti circa al 9.4%, sintomo di una specificità buona ma non perfetta. Tuttavia, durante un'analisi manuale più approfondita del rumore rilevato ho osservato come questo sia *sempre* dovuto al caricamento di alcune delle DLL monitorate da parte di processi diversi da Mimikatz e mai causato da accessi sospetti a lsass.exe. Pertanto, la specificità di quest'ultimo IoC è sicuramente migliore di quella dell'IoC ottenuto combinando i due criteri di detection utilizzati. Complessivamente, ritengo che la bontà degli IoC per questo modulo sia quindi stata, per la maggior parte, confermata dai dati osservati.

Passando invece al modulo **lsadump::dcsync**, ho analizzato un singolo IoC, proposto da [8], consistente in eventi di log caratterizzati dall'ID 4662 e da alcuni specifici valori sospetti. I risultati ottenuti sono osservabili in figura 3. Per quanto riguarda l'affidabilità notiamo come essa sia anche in questo caso molto buona, con un miss rate pari all'1%, potenzialmente derivante da qualche imperfezione nel processo di raccolta dei log. A livello di specificità possiamo invece osservare il limite esposto nel capitolo precedente per questo IoC: a causa del tasso di rumore estremamente elevato – pari al 48.56% – notiamo come l'utilizzo nella pratica di tale strategia di detection possa richiedere l'analisi di una grande quantità di falsi positivi.

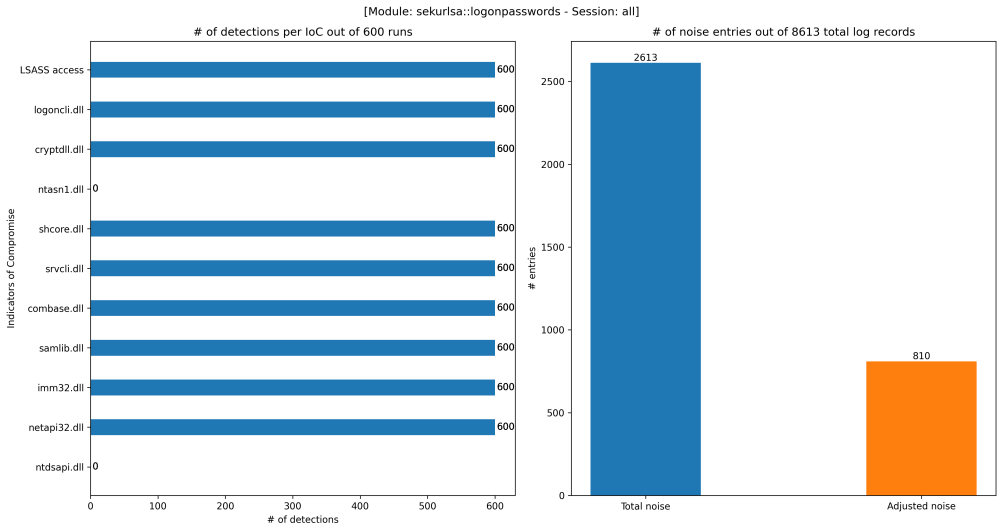

Figura 2: Risultati ottenuti dall'analisi degli IoC noti in letteratura per il modulo **sekurlsa::logonpasswords**.

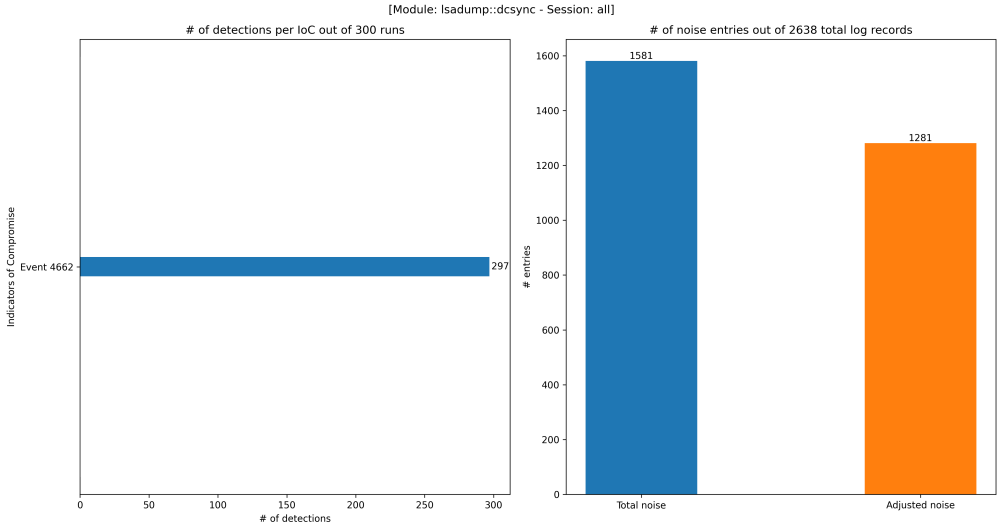

Figura 3: Risultati ottenuti dall'analisi degli IoC noti in letteratura per il modulo **lsadump::dcsync**.

strategy = 'strict'	Firefox	Chrome	Edge	Calc	Writer	WinDBG	x64dbg
sekurlsa::logonpasswords::all	0.89%	1.33%	0.89%	n.a.	n.a.	0.89%	1.33%
lsadump::dcsync::all	2.79%	1.40%	0.47%	n.a.	n.a.	2.79%	2.79%
token::elevate::all	0.70%	1.40%	0.70%	n.a.	n.a.	0.70%	0.70%

strategy = 'lax'	Firefox	Chrome	Edge	Calc	Writer	WinDBG	x64dbg
sekurlsa::logonpasswords::all	5.33%	46.67%	5.33%	n.a.	n.a.	11.56%	32.44%
lsadump::dcsync::all	2.79%	21.40%	2.79%	n.a.	n.a.	7.44%	21.40%
token::elevate::all	0.70%	44.91%	2.46%	n.a.	n.a.	0.70%	12.63%

strategy = 'strict'	Mimikatz #1	Mimikatz #2	Mimikatz #3	Mimikatz #4	Mimikatz #5
sekurlsa::logonpasswords::all	0.89%	0.89%	79.56%	0.89%	0.89%
lsadump::dcsync::all	n.a.	n.a.	100%	45.58%	n.a.
token::elevate::all	0.70%	0.70%	21.75%	n.a.	n.a.

strategy = 'lax'	Mimikatz #1	Mimikatz #2	Mimikatz #3	Mimikatz #4	Mimikatz #5
sekurlsa::logonpasswords::all	100%	84.89%	79.56%	79.56%	80%
lsadump::dcsync::all	n.a.	n.a.	100%	45.58%	n.a.
token::elevate::all	0.70%	0.70%	75.44%	n.a.	n.a.

Tabella 3: Risultati ottenuti dal benchmark delle firme più generiche ricavate per ciascun modulo di Mimikatz analizzato. I colori nella parte inferiore della tabella sono forniti per facilitare al lettore l'associazione tra ciascun modulo di Mimikatz e le istanze di benchmark generate da una sua esecuzione.

Benchmark delle firme generate

I risultati del benchmark finale, effettuato sulle firme più generiche[11] ricavate per ciascun modulo analizzato, sono presentati nella tabella 3.

Osserviamo innanzitutto come, nel caso delle istanze di test derivate da software non malevolo, nessuna delle firme ha generato falsi positivi, commettendo un errore massimo pari al 44.67% nel caso di utilizzo della strategia **lax**. Nonostante ciò, vedremo grazie ai prossimi risultati come, nonostante la tendenza a sovrastimare la presenza di una firma all'interno di un campione, tale strategia sia fortemente preferibile in quanto caratterizzata da una capacità di generalizzazione molto maggiore nel caso di moduli di Mimikatz diversi da quelli utilizzati per la generazione della firma considerata.

Notiamo inoltre come in alcuni casi, denotati dalla dicitura "n.a.", la rilevazione delle firme all'interno dei campioni non è stata possibile in quanto le sequenze di syscall ottenute in seguito alla fase di compressione e filtraggio sono risultate più corte delle firme considerate. In questi casi potremmo implicitamente considerare l'esito della detection *negativo*, in quanto la firma non può effettivamente essere presente all'interno del campione, ma per motivi di chiarezza tali situazioni sono state considerate separatamente dagli altri casi.

In particolare, questa situazione si è *sempre* verificata nel caso di OpenOffice Writer e Calc: a causa di un bug di **drstrace**, entrambi i programmi sono infatti crashati poco dopo l'avvio, senza avere il tempo di generare una quantità di dati sufficiente per l'analisi.

sekurlsa::logonpasswords

Nel caso della strategia **strict**, osserviamo come la firma abbia fornito prestazioni pessime, non riuscendo a effettuare la detection nelle istanze generate dall'esecuzione del modulo associato (#1 e #2). Anche a livello di generalizzazione ad altri moduli i risultati sono scadenti, con la curiosa eccezione – comune a sostanzialmente tutte le firme – dell'istanza #3, generata mediante l'esecuzione del modulo **lsadump::dcsync**.

Al contrario, mediante la strategia **lax**, tale firma ha ottenuto i risultati migliori tra tutte, identificando correttamente l'esecuzione di Mimikatz in *tutte* le istanze testate con una confidenza minima del 79.56%.

lsadump::dcsync

Nel caso della strategia **strict**, la firma ricavata da questo modulo ha presentato risultati *leggermente* migliori della precedente, identificando correttamente l'istanza ad essa associata con una confidenza del 100%. In termini di generalizzazione tuttavia, essa sembra esser stata limitata o dalla sua eccessiva lunghezza o – dualmente – dalla lunghezza ridotta dei campioni disponibili.

[11] Qualora il lettore fosse interessato, i risultati delle firme generate per *tutte* le sessioni sono disponibili all'interno del file **results/all_signatures_results.txt**.

Risultati completamente identici sono stati osservati anche mediante l'utilizzo della strategia **lax**.

token::elevate

Nel caso della strategia **strict** la firma derivata da questo modulo ha ottenuto i risultati peggiori tra tutte, non riuscendo nè a identificare correttamente l'istanza ad esso associata (#4) nè a generalizzare a nessuna delle altre.

Sostanzialmente analoghi sono stati i risultati nel caso della strategia **lax**, se non anche qui per il curioso caso di generalizzazione all'istanza di test generata mediante l'esecuzione di **lsadump::dcsync** (#3).

Le prestazioni particolarmente scadenti di questo modulo potrebbero essere legate al basso numero di esecuzioni utilizzate per la generazione della firma – solo 100 – tuttavia giustificate a livello pratico dall'insolita lentezza riscontrata durante il processo raccolta dei dati mediante **drstrace**.

Conclusioni

Complessivamente, la strategia di detection proposta non ha portato a risultati sufficientemente buoni da giustificarne un possibile utilizzo come alternativa agli IoC tradizionali. In particolare, ritengo che i motivi più probabili alla base di ciò possano essere:

- eccessiva semplicità della procedura di generazione e/o rilevazione delle firme di esecuzione;
- bassa quantità o qualità dei dati raccolti nelle diverse sessioni effettuate.

In entrambi i casi, una possibile estensione del mio lavoro potrebbe pertanto essere uno studio più approfondito delle firme generate e delle cause dietro alle basse prestazioni rilevate, finalizzato all'introduzione di accorgimenti implementativi in grado di migliorare la qualità della detection o alla definizione di strategie migliori per la raccolta dei dati utilizzati per l'analisi.

Bibliografia

[1] *Blog: Mimikatz: The finest in post-exploitation*. Apr. 2021. URL: https://www.cisecurity.org/blog/mimikatz-the-finest-in-post-exploitation/.

[2] Andrew Case. *Memory forensics R&D illustrated: Detecting Mimikatz's skeleton key attack*. URL: https://volatility-labs.blogspot.com/2021/10/memory-forensics-r-illustrated.html.

[3] Neil Fox. *Mimikatz usage & detection*. URL: https://neil-fox.github.io/Mimikatz-usage-&-detection/.

[4] Mohamed El-Hadidi e Marianne Azer. «Detecting Mimikatz in Lateral Movements Using Mutex». In: dic. 2020, pp. 1–6. DOI: 10.1109/ICCES51560.2020.9334643.

[5] Jake Liefer. *Detecting in-memory mimikatz*. Set. 2020. URL: https://sra.io/blog/detecting-in-memory-mimikatz/.

[6] Sean Metcalf. *Unofficial Mimikatz Guide*. Feb. 2018. URL: https://adsecurity.org/?page_id=1821.

[7] *Mimikatz - 2021 threat detection report*. URL: https://redcanary.com/threat-detection-report/threats/mimikatz/.

[8] Brian O'Hara. *Detecting DCSync*. Dic. 2020. URL: https://blog.blacklanternsecurity.com/p/detecting-dcsync.

[9] Oleg Savenko et al. «Dynamic Signature-based Malware Detection Technique Based on API Call Tracing». In: *ICTERI Workshops*. 2019.