A System for Efficiently Hunting for Cyber Threats in Computer Systems Using Threat Intelligence

Peng Gao*, Fei Shao†, Xiaoyuan Liu*, Xusheng Xiao†, Haoyuan Liu*, Zheng Qin‡, Fengyuan Xu‡
Prateek Mittal§, Sanjeev R. Kulkarni§, Dawn Song*
*University of California, Berkeley †Case Western Reserve University
‡National Key Lab for Novel Software Technology, Nanjing University §Princeton University
*{penggao,xiaoyuanliu,hy.liu,dawnsong}@berkeley.edu †{fxs128,xusheng.xiao}@case.edu
‡{qinzheng,fengyuan.xu}@nju.edu.cn §{pmittal,kulkarni}@princeton.edu

***Abstract*—Log-based cyber threat hunting has emerged as an important solution to counter sophisticated cyber attacks. However, existing approaches require non-trivial efforts of manual query construction and have overlooked the rich external knowledge about threat behaviors provided by open-source Cyber Threat Intelligence (OSCTI). To bridge the gap, we build THREATRAPTOR, a system that facilitates cyber threat hunting in computer systems using OSCTI. Built upon mature system auditing frameworks, THREATRAPTOR provides (1) an unsupervised, light-weight, and accurate NLP pipeline that extracts structured threat behaviors from unstructured OSCTI text, (2) a concise and expressive domain-specific query language, TBQL, to hunt for malicious system activities, (3) a query synthesis mechanism that automatically synthesizes a TBQL query from the extracted threat behaviors, and (4) an efficient query execution engine to search the big system audit logging data.**

I. INTRODUCTION

Recent cyber attacks have plagued many high-profile businesses [1]. These attacks often exploit multiple types of vulnerabilities to infiltrate into target systems in multiple stages. To counter these attacks, *ubiquitous system auditing* has emerged as an important approach for monitoring system activities. System auditing collects system-level auditing events about system calls from OS kernel as system audit logs. The collected audit logging data further enables approaches to hunt for cyber threats via query processing [2]–[5].

Cyber threat hunting in enterprises is the process of proactively and iteratively searching for malicious activities in various types of logs, which is critical to early-stage detection. Despite numerous efforts [2], [6], existing approaches, however, require non-trivial efforts of manual query construction and have overlooked the rich external threat knowledge provided by open-source Cyber Threat Intelligence (OSCTI). Hence, the threat hunting process is labor-intensive and error-prone.

OSCTI [7] is a form of evidence-based knowledge and has received growing attention from the community. Commonly, knowledge about threats is presented in a vast number of publicly available OSCTI sources. Structured OSCTI feeds [8] have primarily focused on Indicators of Compromise (IOCs), such as malicious file/process names and IP addresses. Though useful in capturing fragmented views of threats, these disconnected IOCs lack the capability to uncover the complete threat scenario as to how the threat unfolds into multiple steps. In contrast, unstructured OSCTI reports [9] contain more comprehensive threat knowledge. For example, descriptive relationships between IOCs contain knowledge about multi-step threat behaviors (e.g., "read" relationship between two IOCs "/bin/tar" and "/etc/passwd" in Figure 2), which is critical to uncovering the complete threat scenario. Unfortunately, none of the existing approaches provide an automated way to harvest such knowledge and use it for threat hunting.

There are two major challenges for building a system that (1) extracts knowledge about threat behaviors (IOCs and IOC relationships) from unstructured OSCTI reports, and (2) uses the knowledge for threat hunting. First, accurately extracting threat knowledge from natural-language OSCTI text is not trivial. This is due to the presence of massive nuances particular to the security context, such as special characters (e.g., dots, underscores) in IOCs. These nuances limit the performance of most NLP modules (e.g., tokenization). Second, system auditing often produces a huge amount of daily logs (0.5 GB $\sim$ 1 GB for 1 enterprise host [10]), and hence threat hunting is a procedure of "finding a needle in a haystack". Such a big amount of log data poses challenges for solutions to store and query the data efficiently to hunt for malicious activities.

To address these challenges, we build THREATRAPTOR, a system that facilitates threat hunting in computer systems using OSCTI. THREATRAPTOR ($\sim$ 25K LOC) was built upon mature system auditing frameworks for system audit logging data collection and databases for data storage. Particularly, THREATRAPTOR has four novel designs: (1) An unsupervised, light-weight, and accurate NLP pipeline for extracting threat behaviors (IOCs and IOC relations) from OSCTI texts. The pipeline employs a series of techniques (e.g., IOC protection, dependency parsing-based IOC relation extraction) to handle nuances and perform accurate extraction. The extracted IOCs and IOC relations form a *threat behavior graph*, which is amenable to automated processing; (2) A concise and expressive domain-specific query language, *Threat Behavior Query Language (TBQL)*, for querying system audit logging data stored in different database backends. TBQL is a declarative query language that uniquely integrates a series of primitives for threat hunting in computer systems (e.g., system entities, system events, event path patterns, various types of filters); (3) A query synthesis mechanism for automatically synthesizing

2375-026X/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00309

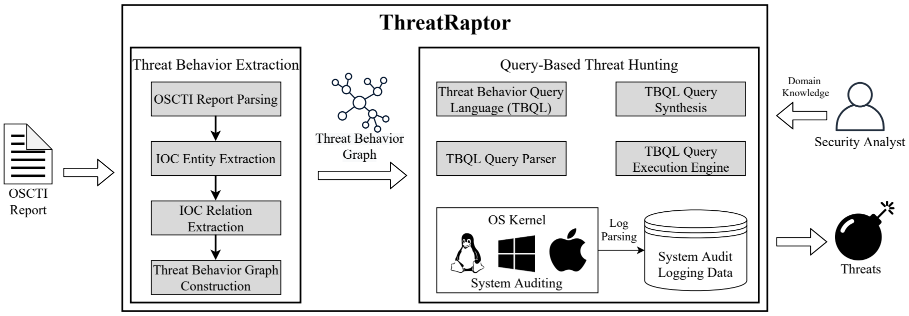


Fig. 1: The architecture of THREATRAPTOR

a TBQL query from the threat behavior graph; (4) A query execution engine for efficiently executing TBQL queries. To the best of our knowledge, THREATRAPTOR is *the first system that bridges OSCTI with system auditing for threat hunting*. For more details, please refer to our full-length paper [11].

Demo video: https://youtu.be/SrcTDQwRF_M

II. THE THREATRAPTOR ARCHITECTURE

Figure 1 shows the architecture of THREATRAPTOR. Given an input OSCTI report, THREATRAPTOR extracts IOCs (e.g., file names, file paths, IPs) and IOC relations, constructs a threat behavior graph, synthesizes a TBQL query, and executes the synthesized query to retrieve the matched system auditing records. Figure 2 shows an example data leakage attack case demonstrating the whole pipeline.

A. Data Collection

System audit logging data records the interactions among system entities as system events. Following the established convention [2], [10], we consider system entities as *files*, *processes*, and *network connections*. We consider a system event as the interaction between two system entities represented as ⟨subject, operation, object⟩. Subjects are processes originating from software applications (e.g., Chrome), and objects can be files, processes, and network connections. We categorize system events into three types according to the types of their object entities: *file events*, *process events*, and *network events*.

THREATRAPTOR leverages a mature system auditing framework, Sysdig, to collect system audit logs from a host. THREATRAPTOR then parses the collected logs into system entities and system events, and extracts critical attributes. Representative entity attributes are: file name, process executable name, src/dst IP, src/dst port. Representative event attributes are: sbj/obj entity ID, operation, start/end time.

B. Data Storage

THREATRAPTOR leverages a relational database, PostgreSQL, and a graph database, Neo4j, for its storage component. Relational databases come with mature indexing mechanisms and are scalable to massive data, which are suitable for queries that involve many joins and constraints. Graph databases represent data as nodes and edges, which are suitable for queries that involve graph pattern search. For PostgreSQL, THREATRAPTOR stores system entities and system events in tables. For Neo4j, THREATRAPTOR stores system entities as nodes and system events as edges. Indexes are created on key attributes to speed up the search. Furthermore, to reduce the data size, THREATRAPTOR leverages the Causality Preserved Reduction technique [10] to merge excessive events between the same pair of entities.

C. Threat Behavior Extraction

THREATRAPTOR employs a specialized NLP pipeline (built upon spaCy) to accurately extract IOCs and IOC relations and construct a threat behavior graph (Algorithm 1).

(1) *Block Segmentation (Line 3) and Sentence Segmentation (Line 6):* We segment an input OSCTI article into natural blocks. We then segment a block into sentences.

(2) *IOC Recognition and IOC Protection (Line 5):* We construct a set of regex rules to recognize various types of IOCs (e.g., file name, file path, IP). Furthermore, we protect the security context by replacing the IOCs with a dummy word (i.e., word "something"). This makes the NLP modules designed for processing general text work well for OSCTI text.

(3) *Dependency Parsing (Line 7):* We construct a dependency tree for each sentence. We then replace the dummy word with the original IOCs in the trees.

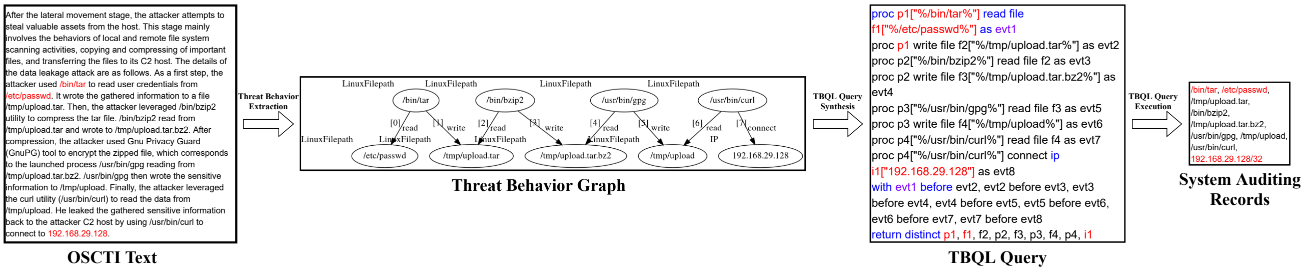
(4) *Tree Annotation (Line 9):* We annotate nodes in the dependency trees whose associated tokens are useful for coreference resolution and relation extraction tasks (e.g., IOCs, candidate IOC relation verbs, pronouns).

(5) *Tree Simplification (Line 10):* We simplify the annotated trees by removing paths without IOC nodes down to the leaves.

(6) *Coreference Resolution (Line 13):* Across all trees of all sentences within a block, we resolve the coreference nodes for the same IOC by checking their POS tags and dependencies, and create connections between the nodes in the trees.

(7) *IOC Scan and Merge (Line 15):* We scan all IOCs in the trees of all blocks, and merge similar ones based on both the character-level overlap and the word vector similarities.

(8) *IOC Relation Extraction (Line 17):* For each dependency tree, we enumerate all pairs of IOCs nodes. Then, for each pair, we check whether they satisfy the subject-object relation by considering their dependency types in the tree. In particular, we consider three parts of their dependency path: one common path from the root to the LCA (Lowest Common Ancestor); two individual paths from the LCA to each of the nodes, and construct a set of dependency type rules to do the checking.



After the lateral movement stage, the attacker attempts to steal valuable assets from the host. This stage mainly involves the behaviors of local and remote file system scanning activities, copying and compressing of important files, and transferring the files to its C2 host. The details of the data leakage attack are as follows. As a first step, the attacker used /bin/tar to read user credentials from /etc/passwd. It wrote the gathered information to a file /tmp/upload.tar. Then, the attacker leveraged /bin/bzip2 utility to compress the tar file. /bin/bzip2 read from /tmp/upload.tar and wrote to /tmp/upload.tar.bz2. After compression, the attacker used Gnu Privacy Guard (GnuPG) tool to encrypt the zipped file, which corresponds to the launched process /usr/bin/gpg reading from /tmp/upload.tar.bz2. /usr/bin/gpg then wrote the sensitive information to /tmp/upload. Finally, the attacker leveraged the curl utility (/usr/bin/curl) to read the data from /tmp/upload. He leaked the gathered sensitive information back to the attacker C2 host by using /usr/bin/curl to connect to 192.168.29.128.

```
proc p1["%/bin/tar%"] read file
f1["%/etc/passwd%"] as evt1
proc p1 write file f2["%/tmp/upload.tar%"] as evt2
proc p2["%/bin/bzip2%"] read file f2 as evt3
proc p2 write file f3["%/tmp/upload.tar.bz2%"] as
evt4
proc p3["%/usr/bin/gpg%"] read file f3 as evt5
proc p3 write file f4["%/tmp/upload%"] as evt6
proc p4["%/usr/bin/curl%"] read file f4 as evt7
proc p4["%/usr/bin/curl%"] connect ip
i1["192.168.29.128"] as evt8
with evt1 before evt2, evt2 before evt3, evt3
before evt4, evt4 before evt5, evt5 before evt6,
evt6 before evt7, evt7 before evt8
return distinct p1, f1, f2, p2, f3, p3, f4, p4, i1
```

/bin/tar, /etc/passwd,
/tmp/upload.tar,
/bin/bzip2,
/tmp/upload.tar.bz2,
/usr/bin/gpg, /tmp/upload,
/usr/bin/curl,
192.168.29.128/32

Fig. 2: An example data leakage attack case demonstrating the whole processing pipeline of ThreatRaptor

Algorithm 1: Threat Behavior Extraction Pipeline

```
   Input  : OSCTI Text: document
   Output: Threat Behavior Graph: graph
1  Initialize all_block_trees;
2  Initialize all_ioc_rels;
3  for block in SegmentBlock(document) do
4      Initialize trees;
5      block ← ProtectIoc(block);
6      for sentence in SegmentSentence(block) do
7          tree ← ParseDependency(sentence);
8          tree ← RemoveIocProtection(tree);
9          tree ← AnnotateTree(tree);
10         tree ← SimplifyTree(tree);
11         Add tree to trees;
12     for tree in trees do
13         tree ← ResolveCoref(tree, trees);
14     Add all tree in trees to all_block_trees;
15 all_iocs ← ScanMergeIoc(all_block_trees);
16 for tree in trees do
17     ioc_rels ← ExtractIocRelation(tree, trees, all_iocs);
18     Add ioc_rels to all_ioc_rels;
19 graph ← ConstructGraph(all_iocs, all_ioc_rels);
```

Next, for the pair that passes the checking, we extract its relation verb by first scanning all the annotated candidate verbs in the aforementioned three parts of dependency path, and then selecting the one that is closest to the object IOC node. The candidate IOC node pair and the selected verb (after lemmatization) then form the final IOC entity-relation triplet.

(10) *Threat Behavior Graph Construction (Line 19):* We iterate over all IOC entity-relation triplets sorted by the occurrence offset of the relation verb in OSCTI text, and construct a threat behavior graph. Each edge in the graph is associated with a sequence number, indicating the step order.

D. Threat Behavior Query Language (TBQL)

ThreatRaptor provides a domain-specific language, TBQL (built upon ANTLR 4), to query system audit logging data. Compared to general-purpose query languages (e.g., SQL, Cypher) that are low-level and verbose, TBQL treats system entities and events as first-class citizens and provides primitives to easily specify multi-step system activities.

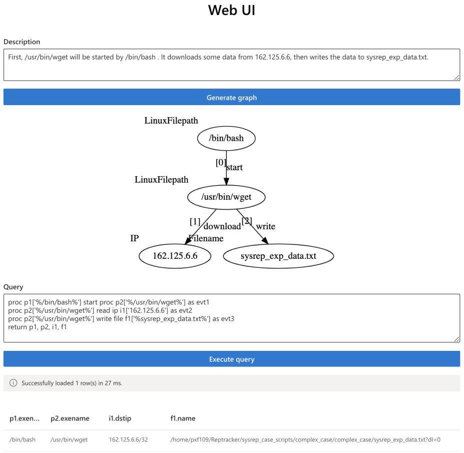
The basic *event pattern syntax* of TBQL specifies one or more system event patterns in the format of ⟨subject, operation, object⟩, with optional filters on the temporal and attribute relationships between event patterns. System entities have explicit types and identifiers, with optional filters on the entity attributes. Operators (e.g., logical, comparison) are supported in event operations and attribute filters to form complex expressions. Optional time windows can be specified for event patterns to constrain the search.

Figure 2 shows an example synthesized TBQL query in this syntax. Eight event patterns are declared (**`evt1 - evt8`**), with entity types, identifiers, and attribute filters. The `with` clause specifies the temporal orders of events (i.e., temporal relationships). Besides, several syntactic sugars are adopted to make the query concise: (1) default attribute names are omitted in the event patterns and the `return` clause, which will be inferred during query execution. We select the most commonly used attributes in security analysis as default attributes: "name" for files, "exename" for processes, and "dstip" for network connections. For example, **`proc p1["%/bin/tar%"]`** will be inferred as **`proc p1[exename = "%/bin/tar%"]`** and **`return p1`** will be inferred as **`return p1.exename`**; (2) an entity ID is used in multiple event patterns, which means that the referred entities are the same. For example, **`p1`** is used in both **`evt1`** and **`evt2`**, which is equivalent to an attribute relationship between the two event patterns, i.e., **`evt1.srcid = evt2.srcid`**.

Besides the basic syntax, ThreatRaptor provides an advanced syntax that specifies *variable-length paths of system event patterns*. This syntax is particularly useful when doing query synthesis: in some cases, an edge in the threat behavior graph may correspond to a path of system events in system audit logging data. This happens often when intermediate processes are forked to chain system events, but are omitted in the OSCTI text by the human writer. For example, **`proc p ~>[read] file f`** specifies a path of arbitrary length from a process entity **`p`** to a file entity **`f`**. The operation type of the final hop is **`read`**. **`proc p ~>(2~4)[read] file f`** furthers specifies the minimum and maximum lengths of the path. More language features are illustrated in our demo video.

E. TBQL Query Synthesis

ThreatRaptor provides a query synthesis mechanism that automatically synthesizes a TBQL query from the threat behavior graph. The synthesis starts with a screening to filter out nodes (and connected edges) in the threat behavior graph whose associated IOC types are not currently captured by the system auditing component. Then, for each remaining edge, ThreatRaptor maps its associated IOC relation to the TBQL operation type using a set of rules (e.g., the "download" relation between two "Filepath" IOCs will be mapped to the "write" operation in TBQL, indicating a process writes data to a file). Next, ThreatRaptor synthesizes the subject/object entity from the source/sink node, and synthesizes



Description

First, /usr/bin/wget will be started by /bin/bash . It downloads some data from 162.125.6.6, then writes the data to sysrep_exp_data.txt.

Query

```
proc p1['%/bin/bash%'] start proc p2['%/usr/bin/wget%'] as evt1
proc p2['%/usr/bin/wget%'] read ip i1['162.125.6.6'] as evt2
proc p2['%/usr/bin/wget%'] write file f1['%sysrep_exp_data.txt%'] as evt3
return p1, p2, i1, f1
```

ⓘ Successfully loaded 1 row(s) in 27 ms.

p1.exen...	p2.exename	i1.dstip	f1.name
/bin/bash	/usr/bin/wget	162.125.6.6/32	/home/pxf109/Reptracker/sysrep_case_scripts/complex_case/complex_case/sysrep_exp_data.txt?dl=0

Fig. 3: The web UI of ThreatRaptor

an event pattern by connecting the entities with the operation. ThreatRaptor then synthesizes the temporal relationships of the event patterns in the `with` clause based on the sequence numbers of the corresponding edges. Finally, ThreatRaptor synthesizes the `return` clause by appending all entity IDs. In addition to the default synthesis plan, ThreatRaptor supports user-defined plans to synthesize other patterns (e.g., path patterns) and attributes (e.g., time window).

F. TBQL Query Execution

To execute a TBQL query with multiple patterns, ThreatRaptor compiles each pattern into a semantically equivalent SQL or Cypher data query, and schedules the execution of these data queries in different database backends. Specifically, for an event pattern, ThreatRaptor compiles it into a SQL data query which joins entity tables with event table. For a variable-length event path pattern, since it is difficult to perform graph pattern search using SQL, ThreatRaptor compiles it into a Cypher data query by leveraging Cypher's path pattern syntax.

For each pattern, ThreatRaptor computes a pruning score by counting the number of constraints declared; a pattern with more constraints has a higher score. For a variable-length event path pattern, ThreatRaptor additionally considers the path length; a pattern with a smaller maximum path length has a higher score. Then, when scheduling the execution of the data queries, ThreatRaptor considers both the pruning scores and the pattern dependencies: if two patterns are connected by the same system entity, ThreatRaptor will first execute the data query whose associated pattern has a higher pruning score, and then use the execution results to constrain the execution of the other data query (by adding filters). This way, complex TBQL queries can be efficiently executed in different database backends seamlessly.

III. Demonstration Outline

We deployed ThreatRaptor on a server and built a web UI (Figure 3). In our demo, we aim to show the complete usage scenario of ThreatRaptor. We perform two multi-step intrusive attacks on the deployed server and construct attack descriptions according to the way the attacks were performed. Constructed based on CVE [12], these two attacks exploit system vulnerabilities and exfiltrate sensitive data:

- *Password Cracking After Shellshock Penetration:* The attacker penetrates into the victim host (i.e., the deployed server) by exploiting the Shellshock vulnerability. After the penetration, the attacker first connects to cloud services (Dropbox) and downloads an image where C2 (Command and Control) server's IP address is encoded in the EXIF metadata. Based on the IP address, the attacker downloads a password cracker from the C2 server to the victim host. The attacker then runs the password cracker against password shadow files to extract clear text.
- *Data Leakage After Shellshock Penetration:* The attacker attempts to steal all the valuable assets from the victim host. The attacker scans the file system, scrapes files into a single compressed file, and transfers it back to the C2 server.

When we perform the attacks, the server continues to resume its routine tasks to emulate the real-world deployment, where benign system activities and malicious system activities co-exist. We use ThreatRaptor to hunt for malicious system activities by feeding the attack descriptions to the system, which in turn synthesizes TBQL queries and executes the synthesized queries over the collected data. The audience will have the option to conduct the attacks and perform threat hunting through ThreatRaptor's web UI.

IV. Conclusion

We have presented ThreatRaptor, a novel system that facilitates threat hunting in computer systems using OSCTI.

Acknowledgement. This work was supported in part by DARPA N66001-15-C-4066 and the CLTC (Center for Long-Term Cybersecurity).

References

[1] "The Equifax Data Breach," https://www.ftc.gov/equifax-data-breach.

[2] P. Gao, X. Xiao, Z. Li, F. Xu, S. R. Kulkarni, and P. Mittal, "AIQL: Enabling efficient attack investigation from system monitoring data," in *USENIX ATC*, 2018.

[3] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "SAQL: A stream-based query system for real-time abnormal system behavior detection," in *USENIX Security*, 2018.

[4] P. Gao, X. Xiao, Z. Li, K. Jee, F. Xu, S. R. Kulkarni, and P. Mittal, "A query system for efficiently investigating complex attack behaviors for enterprise security," in *VLDB*, 2019.

[5] P. Gao, X. Xiao, D. Li, K. Jee, H. Chen, S. R. Kulkarni, and P. Mittal, "Querying streaming system monitoring data for enterprise system anomaly detection," in *ICDE*, 2020.

[6] "Splunk Search Processing Language," https://www.splunk.com/en_us/resources/search-processing-language.html.

[7] "Open Source Threat Intelligence Feeds," https://www.senki.org/operators-security-toolkit/open-source-threat-intelligence-feeds/.

[8] "Structured Threat Information eXpression," http://stixproject.github.io/.

[9] "SecureList," https://securelist.com/.

[10] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *CCS*, 2016.

[11] P. Gao, F. Shao, X. Liu, X. Xiao, Z. Qin, F. Xu, P. Mittal, S. R. Kulkarni, and D. Song, "Enabling efficient cyber threat hunting with cyber threat intelligence," in *ICDE*, 2021.

[12] "Common Vulnerabilities and Exposures," https://cve.mitre.org/.