Diagnosing Advanced Persistent Threats: A Position Paper

Rui Abreu and **Danny Bobrow** and **Hoda Eldardiry** and **Alexander Feldman** and **John Hanley** and **Tomonori Honda** and **Johan de Kleer** and **Alexandre Perez**
Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto, CA 94304, USA
{rui,bobrow,hoda.eldardiry,afeldman,john.hanley,tomo.honda,dekleer,aperez}@parc.com

Dave Archer and **David Burke**
Galois, Inc.
421 SW 6th Avenue, Suite 300
Portland, OR 97204, USA
{dwa,davidb}@galois.com

Abstract

When a computer system is hacked, analyzing the root-cause (for example entry-point of penetration) is a diagnostic process. An audit trail, as defined in the National Information Assurance Glossary, is a security-relevant chronological (set of) record(s), and/or destination and source of records that provide evidence of the sequence of activities that have affected, at any time, a specific operation, procedure, or event. After detecting an intrusion, system administrators manually analyze audit trails to both isolate the root-cause and perform damage impact assessment of the attack. Due to the sheer volume of information and low-level activities in the audit trails, this task is rather cumbersome and time intensive. In this position paper, we discuss our ideas to automate the analysis of audit trails using machine learning and model-based reasoning techniques. Our approach classifies audit trails into the high-level activities they represent, and then reasons about those activities and their threat potential in real-time and forensically. We argue that, by using the outcome of this reasoning to explain complex evidence of malicious behavior, we are equipping system administrators with the proper tools to promptly react to, stop, and mitigate attacks.

1 Introduction

Today, enterprise system and network behaviors are typically "opaque": stakeholders lack the ability to assert causal linkages in running code, except in very simple cases. At best, event logs and audit trails can offer some partial information on temporally and spatially localized events as seen from the viewpoint of individual applications. Thus current techniques give operators little system-wide situational awareness, nor any viewpoint informed by a long-term perspective. Adversaries have taken advantage of this opacity by adopting a strategy of persistent, low-observability operation from inside the system, hiding effectively through the use of long causal chains of system and application code. We call such adversaries *advanced persistent threats*, or APTs.

To address current limitations, this position paper discusses a technique that aims to track causality across the enterprise and over extended periods of time, identify subtle causal chains that represent malicious behavior, localize the code at the roots of such behavior, trace the effects of other malicious actions descended from those roots, and make recommendations on how to mitigate those effects. By doing so, the proposed approach aims to enable stakeholders to understand and manage the activities going on in their networks. The technique exploits both current and novel forms of local causality to construct higher-level observations, long-term causality in system information flow. We propose to use a machine learning approach to classify segments of low-level events by the activities they represent, and reasons over these activities, prioritizing candidate activities for investigation. The diagnostic engine investigates these candidates looking for patterns that may represent the presence of APTs. Using pre-defined security policies and related mitigations, the approach explains discovered APTs and recommends appropriate mitigations to operators. We plan to leverage models of APT and normal business logic behavior to diagnose such threats. Note that the technique is not constrained by availability of human analysts, but can benefit by human-on-the-loop assistance.

The approach discussed in the paper will offer unprecedented capability for *observation* of long-term, subtle system-wide activity by automatically constructing such global, long-term causality observations. The ability to automatically classify causal chains of events in terms of abstractions such as activities, will provide operators with a unique capability to *orient* to long-term, system-wide evidence of possible threats. The diagnostic engine will provide a unique capability to identify whether groups of such activities likely represent active threats, making it easier for operators to *decide* whether long-term threats are active, and where they originate, even before those threats are identified by other means. Thus, the approach will pave the way for the first automated, long-horizon, continuously operating system-wide support for an effective defender Observe, Orient, Decide, and Act (OODA) loop.

2 Running Example

The methods proposed in this article are illustrated on a *realistic* running example. The attackers in this example use sophisticated and recently discovered exploits to gain access to the victim's resources. The attack is remote and *does not* require social engineering or opening a malicious email attachment. The methods that we propose, however, are not limited to this class of attacks.

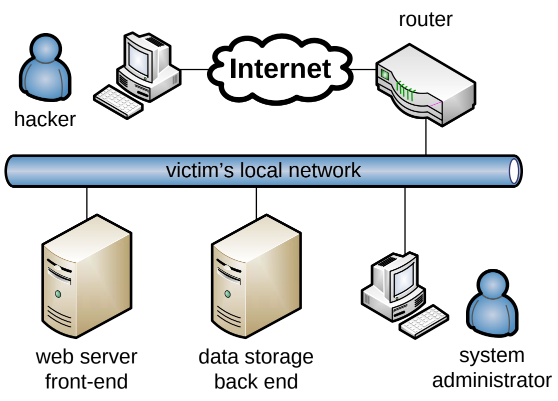

Figure 1: Network topology for the attack

The network topology used for our running example is shown in figure 1. The attack is executed over several days. It starts by (1) compromising the web server front-end, followed by (2) a reconnaissance phase and (3) compromising the data storage back end and ultimately extracting and modifying sensitive information belonging to the victim.

Both the front-end and the back end in this example run unpatched UBUNTU 13.1 LINUX OS on an INTEL® SANDY BRIDGE™ architecture.

What follows is a detailed chronology of the events:

1. The attacker uses the APACHE `httpd` server, a `cgi-bin` script, and the SHELLSHOCK vulnerability (GNU `bash` exploit registered in the Common Vulnerabilities and Exposures database as CVE 2014-6271 (see `https://nvd.nist.gov/`) to gain remote shell access to the victim's front-end. It is now possible for the attacker to execute processes on the front-end as the non-privileged user `www-data`.
2. The attacker notices that the front-end is running an unpatched UBUNTU LINUX OS version 13.1. The attacker uses the `nc` Linux utility to copy an exploit for obtaining root privileges. The particular exploit that the attacker uses utilizes the x32 `recvmmsg()` kernel vulnerability registered in the Common Vulnerabilities and Exposures (CVE) database as CVE 2014-0038. After running the copied binary for a few minutes the attacker gains root access to the front-end host.
3. The attacker installs a root-kit utility that intercepts all input to ssh;
4. A system administrator uses the compromised ssh to connect to the back-end revealing his back-end password to the attacker;
5. The attacker uses the compromised front-end to bypass firewalls and uses the newly acquired back-end administrator's password to access the back-end;
6. The attacker uses a file-tree traversing utility on the back-end that collects sensitive data and consolidates it in an archive file;
7. The attacker sends the archive file to a third-party hijacked computer for analysis.

3 Auditing and Instrumentation

Almost all computing systems of sufficiently high-level (with the exception of some embedded systems) leave detailed logs of all system and application activities. Many UNIX variants such as LINUX log via the `syslog` daemon, while WINDOWS™ uses the event log service. In addition to the usual logging mechanisms, there is a multitude of projects related to secure and detailed auditing. An audit log is more detailed trail of any security or computation-related activity such as file or RAM access, system calls, etc.

Depending on the level of security we would like to provide, there are several methods for collecting input security-related information. On one extreme, it is possible to use the existing log files. On the other extreme there are applications for collecting detailed information about the application execution. One such approach [1] runs the processes of interest through a debugger and logs every memory read and write access.

It is also possible to automatically inject logging calls in the source files before compiling them, allowing us to have static or dynamic logging or a combination of the two. Log and audit information can be signed, encrypted and sent in real-time to a remote server to make system tampering and activity-hiding more difficult. All these configuration decisions impose different trade-offs in security versus computational and RAM load [2] and depend on the organizational context.

```
⋮

front_end.secure_access_log:11.239.64.213 - [22/Apr/2014
06:30:24 +0200] "GET /cgi-bin/test.cgi HTTP/1.1" 401 381

⋮

front_end.rsyslogd.log:recvmsg(3, msg_name(0) =
NULL, msg_iov(1) = ["29/Apr/2014 22:15:49 ...", 8096],
msg_controllen = 0, msg_flags = MSG_CTRUNC,
MSG_DONTWAIT) = 29

⋮

back_end:auditctl:type = SYSCALL msg = au-
dit(1310392408.506:36): arch = c000003e syscall = 2
success = yes exit = 3 a0 = 7fff2ce9471d a1 = 0 a2 = 61f768
a3 = 7fff2ce92a20 items = 1 ppid = 20478 pid = 21013 auid
= 1000 uid = 0 gid = 0 euid = 0 suid = 0 fsuid = 0 egid = 0
sgid = 0 fsgid = 0 ses = 1 comm = "grep" exe = "/bin/grep"

⋮
```

Figure 2: Part of log files related to the attack from the running example

Figure 2 shows part of the logs collected for our running example. The first entry is when the attacker exploits the SHELLSHOCK vulnerability through a CGI script of the web server. The second entry shows syslog `strace`-like message resulting from the kernel escalation. Finally, the attacker uses the `grep` command on the back-end server to search for sensitive information and the call is recorded by the audit system.

It is often the case that the raw system and security log files are preprocessed and initial causal links are computed. If we trace the `exec`, `fork`, and `join` POSIX system calls, for example, it is possible to add graph-like structure to the log files computing provenance graphs. Another method for computing local causal links is to consider shared resources, e.g., two threads reading and writing the same memory address [1].

4 Activity Classification

The Activity Classifier continuously annotates audit trails with semantic tags describing the higher-order activity they represent. For example, 'remote shell access', 'remote file overwrite', and 'intra-network data query' are possible activity tags. These tags are used by the APT Diagnostics Engine to enable higher-order reasoning about related activities, and to prioritize activities for possible investigation.

4.1 Hierarchical semantic annotation of audit trails

A key challenge in abstracting low-level events into higher-order activity patterns that can be reasoned about efficiently is that such patterns can be described at multiple levels of semantic abstraction, all of which may be useful in threat analysis. Indeed, higher-order abstractions may be composed of lower-order abstractions that are in turn abstractions of low-level events. For example, a sequential set of logged *events* such as 'browser forking bash', 'bash initiating Netcat', and 'Netcat listening to new port', might be abstracted as the *activity* 'remote shell access'. The set of activities, 'remote shell access', and 'escalation of privilege' can be abstracted as the activity 'remote **root** shell access'.

We approach activity annotation as a supervised learning problem that uses classification techniques to generate activity tags for audit trails. Table 1 shows multiple levels of activity classifications for the above APT example.

Table 1 represents one possible classification-enriched audit trail for such an APT. There can be many relatively small variations. For example, obscuring the password file could be done using other programs. A single classifier only allows for a single level of abstraction, and a single leap from low-level events to very abstract activities (for example, from 'bash execute perl' level to 'extracting modified file' level) will have higher error caused by these additional variations.

To obtain several layers of abstraction for reasoning over, and thus reduce overall error in classification, we use a *multi-level learning* strategy that models information at multiple levels of semantic abstraction using multiple classifiers. Each classifier solves the problem at one abstraction level, by mapping from a lower-level (fine) feature space to the next higher-level conceptual (coarse) feature space.

The activity classifier rely on both a vocabulary of activities and a library of patterns describing these activities that will be initially defined manually. This vocabulary and pattern set reside in a Knowledge Base.

In our training approach, results from training lower level classifiers are used as training data for higher level classifiers. In this way, we coherently train all classifiers by preventing higher-level classifiers from being trained with patterns that will never be generated by their lower-level precursors. We use an ensemble learning approach to achieve accurate classification. This involves stacking together both bagged and boosted models to reduce both variance and bias error components [3]. The classification algorithm will be trained using an online-learning technique and integrated within an Active Learning Framework to improve classification of atypical behaviors.

Generating Training Data for Classification To build the initial classifier, training data is generated using two methods. First, an actual deployed system is used to collect normal behavior data, and a Subject Matter Expert manually labels it. Second, a testing platform is used to generate data in a controlled environment, particularly platform dependent vulnerability-related behavior. In addition, to generate new training data of previously unknown behavior, we use an Active Learning framework as described in Section 5.

Table 1: Sample classification problem for running example

Activity 1 *Remote Shell Access* Shell Shock	Activity 2 *Remote File Overwrite* Trojan Installation	Activity 3 *Modified File Download* Password Exfiltration
Browser (Port 80) fork bash	Netcat listen to Port 8443	Netcat listen to Port 8443
bash fork Netcat	Port 8443 receive binary file	Port 8443 fork bash
Netcat listen to port 8080	binary file overwrites libns.so	bash execute perl
		Perl overwrite /tmp/stolen_pw
		Port 8443 send /tmp/stolen_pw

5 Prioritizer

As the Activity Classifier annotates audit trails with activity descriptors, the two (parallel) next steps in our workflow are to 1) prioritize potential threats to be referred to the Diagnostic Engine (see Section 6) for investigation, and 2) prioritize emergent activities that (after suitable review and labeling) are added to the activity classifier training data. This module prioritizes activities by *threat severity* and *confidence level*. This prioritization process presents three key challenges.

5.1 Threat-based rank-annotation of activities

One challenge in ranking activities according to their threat potential is the complex (and dynamic) notion of what constitutes a threat. Rankings based on matching to known prior threats is necessary, but not sufficient. An ideal ranking approach should take known threats into account, while also proactively considering the unknown threat potential of new kinds of activities. Another such challenge is that risk may be assessed at various levels of activity abstraction, requiring that overall ranking must be computed by aggregating risk assessments at multiple abstraction levels.

We implement two ranking approaches: a *supervised* ranker based on previously known threats and an *unsupervised* ranker that considers unknown potential threats.

Supervised ranking using APT classification to catch known threats. The goal of APT classification is to provide the diagnostic engine with critical APT related information such as APT Phase, severity of attack, and confidence level associated with APT tagging for threat prioritization. Since the audit trails are annotated hierarchically into different granularity of actions, multiple classifiers will be built to consider each hierarchical level separately. APT classifiers are used to identify entities that are likely to be instances of known threats or phases of an APT attack. Two types of classifiers are used. The first classifier is hand-coded and the second classifier is learned from training data.

The hand-coded classifier is designed to have high precision, using hand-coded rules, mirroring SIEM and IDS systems. Entities tagged by this classifier are given the highest priority for investigation. The second classifier, which is learned from training data, will provide higher recall at the cost of precision. Activities are ranked according to their threat level by aggregating a severity measure (determined by classified threat type) and a confidence measure. We complement the initial set of training data to calibrate our classifiers by using an Active Learning Framework, which focuses on improving the classification algorithm through occasional manual labeling of the most critical activities in the audit trails.

Unsupervised ranking using normalcy characterization to catch unknown threats. The second component of the prioritizer is a set of unsupervised *normalcy rankers*, which rank entities based on their statistical "normalcy". Activities identified as *unusual* will be fed to the Active Learning framework to check if any of them are "unknown" APT activities. This provides a mechanism for detecting "unknown" threats while also providing feedback to improve the APT classifier.

5.2 Combining Multiple Rankings

One of the key issues with combining the outputs of multiple risk ranking is dealing with two-dimensional risk (severity, confidence) scores that may be on very different scales. A diverse set of score normalization techniques have been proposed [4; 5; 6] to deal with this issue, but no single technique has been found to be superior over all the others. An alternative to combining scores is to combine rankings [7]. Although converting scores to rankings does lose information, it remains an open question if the loss in information is compensated for by the convenience of working with the common scale of rankings.

We will develop combination techniques for weighted risk rankings based on probabilistic rank aggregation methods. This approach builds on our own work [8] that shows the robustness of the weighted ranking approach. We also build on principled methods for combining ranking data found in the statistics and information retrieval literature.

Traditionally, the goal of rank aggregation [9; 10] is to combine a set of rankings of the same candidates into a single consensus ranking that is "better" than the individual rankings. We extend the traditional approach to accommodate the specific context of weighted risk ranking. First, unreliable rankers will be identified and either ignored or down-weighted, lest their rankings decrease the quality of the overall consensus [7; 10]. Second, we will discount excessive correlation among rankers, so that a set of highly redundant rankers do not completely outweigh the contribution of other alternative rankings. To address these two issues, we will associate a probabilistic latent variable Z_i with the i'th entity of interest, which indicates whether the entity is anomalous or normal. Then, we will build a probabilistic model that allows us to infer the posterior distribution over the Z_i based on the observed rankings produced by each of the input weighted risk rankings. This posterior probability of Z_i being normal will then be used as the weighted risk rank. Our model will make the following assumptions to account for both unreliable and correlated rankers: 1) Anomalies are ranked lower than all normal instances and these ranks tend to be concentrated near the lower rankings of the provided weighted risk rankings, and 2) Normal data instances tend to be uniformly distributed near the higher rankings of the weighted risk rankings.

There are various ways to build a probabilistic model that reflects the above assumptions and allows for the inference of the Z_i variables through Expectation-Maximization [11]. In addition to these assumptions, we will explore allowing other factors to influence the latent Z_i variables, such as features of the entities as well as feedback provided by an expert analysts.

6 Diagnosis

We view the problem of detecting, isolating, and explaining complex APT campaigns behavior from rich activity data is a diagnostic problem. We will use an AI-based diagnostic reasoning to guide the global search for possible vulnerabilities that enabled the breach. Model-based diagnosis (MBD) [12] is a particularly compelling approach as it supports reasoning over complex causal networks (for example, having multiple conjunctions, disjunctions, and negations) and identifies often subtle combinations of root causes of the symptoms (the breach).

6.1 An MBD approach for APT detection and isolation: Motivation

Attack detection and isolation are two distinct challenges. Often diagnostic approaches use separate models for detection and isolation [13]. MBD however uses a *single model*, to combine these two reasonings. The security model contains both part of the security policy (that communicating with certain blacklisted hosts may indicate an information leak) and information about the *possible* locations and consequence of a vulnerability (a privilege escalation may lead to an information leak). The security model also contains abstract security constraints such as if a process requires authentication, a password must be read and compared against.

The diagnostic approach takes into consideration the bootstrapping of an APT which we consider the root-cause of the attack. What enables a successful APT is either a *combination* of software component vulnerabilities or the combined use of social engineering and insufficiency of the organizational security policies. We use MBD for computing *the set of simultaneously exploited vulnerabilities* that allowed the deployment of the APT. Computing such explanations is possible because MBD reasons in terms of multiple-faults [14]. In our running example this set would include both the fact the the web server has been exploited due to the Shellshock vulnerability and that a the attacker gained privileged access on the front-end due to the use of the X64_32 escalation vulnerability.

The abstract security model is used to gather information about types of attacks the system is vulnerable to, and to aid deciding the set of actions required to stop an APT campaign (policy enforcement). Various heuristics exist to find the set of meaningful diagnosis candidates. As an example, one might be interested in the minimal set of actions to stop the attack [15; 16] or select those candidates that capture significant probability mass [17]. In the rest of this section, for illustration purposes, we use minimality as the heuristic of interest. MBD is the right tool for dealing with computation of diagnosis candidates as it offers several ways to address the modeling and computational complexity [18; 19].

6.2 Detection and Isolation of Attacks from Abstract Security Model and Sensor Data

The abstract security model provides an abstraction mechanism that is originally missing in the audit trails. More precisely what is not in the audit trails and what is in the security model is how to connect (possibly disconnected) activities for the purpose of global reasoning. The abstract security model and the sensor data collected from the audit trails are provided as inputs to an MBD algorithms that performs the high-level reasoning about possible vulnerabilities and attacks similar to what a human security officer would do.

The information in the "raw" audit trails is of too high fidelity [2] and low abstraction to be used by a "crude" security model. That is the reason the diagnostic engine needs the machine learning module to temporally and spatially group nodes in the audit trails and to provide semantically rich variable/value sensor data about actions, suitable for MBD. Notice that in this process, the audit trail structure is translated to semantic categories, i.e., the diagnostic engine receives as observations time-series of sensed actions.

The listing that follows next shows an abstract security model for the running example in the LYDIA language [20]. This bears some resemblance to PROLOG, except that LYDIA is a language for model-based diagnosis of logical circuits while PROLOG is for Horn-style reasoning. The use of LYDIA is for illustration purposes only, in reality computer systems can be much more easily modeled as state machines. There is a significant body of literature dealing with diagnosis of discrete-event systems [21; 22; 23], to name just a few.

```
system front_end (bool know_root_password)
{
    bool httpd_shell_vuln ; //  vulnerability
    bool buffer_overflow_vuln ; //  vulnerability
    bool escalation_vuln ; //  vulnerability

    bool httpd_shell ;
    bool root_shell ;
    bool leak_passwd;

    // weak−fault models
    if (! httpd_shell_vuln ) { //  if healthy
        ! httpd_shell ; //  forbid shells via httpd
    }

    if (! escalation_vuln ) { //  if healthy
        ! root_shell ; //  no root shell is possible
    }

    if (! buffer_overflow_vuln ) { //  if healthy
        !leak_passwd; //  passwords don't leak
    }

    bool access_passwd;
    attribute observable (access_passwd) = true;

    !access_passwd => !leak_passwd;

    /**
     * Knowing the root password can be explained
     * by a root shell (for example there is a
     * password sniffer ).
     */
    know_root_password =>
    (( httpd_shell || leak_passwd) && root_shell );
}

system back_end(bool know_root_password)
{
   bool comm;
   attribute observable (comm) = true;

   /**
    * Normal users can only communicate with a
    * list of permitted hosts.
    */
   if (!know_root_password) {
        comm == true;
   }
}

system main()
{
    bool know_root_password;

    system front_end fe (know_root_password);
    system back_end be(know_root_password);
}
```

Lydia translates the model to an internal propositional logic formula. Part of this internal representation is shown in figure 3, which uses the standard VLSI [24] notation to denote AND-gates, OR-gates, and NOT-gates. Wires are labeled with variable names. Boolean circuits (matching propositional logic), however, have limited expressiveness and modeling secu-

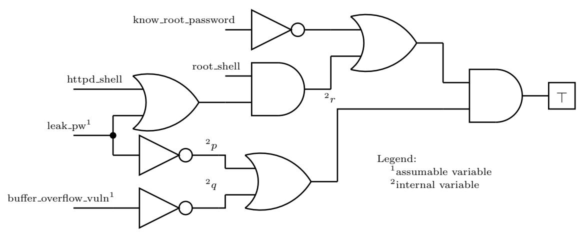

Figure 3: Part of the abstract security model for the running example

rity constraints in it is notoriously difficult, hence we plan to create or use specialized modal logic similar to the one proposed in [25].

Notice that the format of the Boolean circuit shown in figure 3 is very close to the one used in Truth Maintenance System (TMS) [26]. The only assumable variable in figure 3 is `buffer_overflow_vuln` and its default value is false (i.e., there is no buffer overflow vulnerability in the web server process).

We next show how a reasoning engine can discover a conflict through forward and backward propagation. Looking at figure 3, it is clear that r must be true because it is an input to an AND-gate whose output is set to true. Therefore either p or q (or both) must be true. This means that either `buffer_overflow_vuln` or `leak_pw` must be false. If we say that `leak_pw` is assumed to be true (measured or otherwise inferred), then `leak_pw` and `buffer_overflow_vuln` are together part of a conflict. It means that the reasoning engine has to change one of them to resolve the contradiction.

Based on the observation from our running example and a TMS constructed from the security model shown in figure 3, the hitting set algorithm computes two possible diagnostic hypotheses: (1) the attacker gained a shell access through a web-server vulnerability *and* the attacker performed privilege escalation *or* (2) the attacker injected binary code through a buffer overflow *and* the attacker performed privilege escalation.

If we use Lydia to compute the set of diagnoses for the running example, we get the following two (ambiguous) diagnoses for the root-cause of the penetration:

```
$ lydia example.lm example.obs
d1 = { fe.escalation_vuln,
       fe.httpd_shell_vuln }
d2 = { fe.buffer_overflow_vuln,
       fe.escalation_vuln }
```

MBD uses probabilities to computes a sequence of possible diagnoses ordered by likelihood. This probability can be used for many purposes: decide which diagnosis is more likely to be the true fault explanation, whether there is the need for consider further evidence from the logs or limit the number of diagnoses that need to be identified. Many policies exist to compute these probabilities [27; 28].

For illustration purposes we consider that the diagnoses for the running example are ambiguous. Before we discuss methods for dealing with this ambiguity, we address the major research challenge of model generation.

6.3 Model Generation

The abstract vulnerability model can either be constructed manually or semi-automatically. The challenge with modeling is that an APT campaign generally exploits unknown vulnerabilities. Therefore, our approach to address this issue is to construct the model which captures expected behavior (*known goods*) of the system. Starting from generic parameterized vulnerability models and security objectives, the abstract vulnerability model can be extended with information related to known vulnerabilities (*known bads*).

Generating the model can be done either manually or semi-automatically. We will explore venues to generate this model manually, which requires significant knowledge about potential security vulnerabilities, while being error prone and not detailed enough. Amongst company specific requirements, we envisage the abstract vulnerability model to capture the most common attacks that target software systems, as described in the Common Attack Pattern Enumeration and Classification (CAPEC[1]). The comprehensive list of known attacks has been designed to better understand the perspective of an attacker exploiting the vulnerabilities and, from this knowledge, devise appropriate defenses.

As modeling is challenging, we propose to explore semi-automatic approaches to construct models. The semi-automatic method is suitable to addressing the modeling because in security, similarly to diagnosis, there is (1) component models and (2) structure. While it is difficult to automate the building of component models (this may even require natural language parsing of databases such as CAPEC), it is feasible to capture diagnosis-oriented information from structure (physical networking or network communication).

Yet another approach to semi-automatically generate the model is to learn it from executions of the system (e.g., during regression testing, just before deployment). This approach to system modeling is inspired by the work in automatic software debugging work [29], where modeling of program behavior is done in terms of abstraction of program traces – known as spectra [30], abstracting from modeling specific components and data dependencies

The outlined approaches to construct the abstract vulnerability model entail different costs and diagnostic accuracies. As expected, manually building the model is the most expensive one. Note that building the model is a time-consuming and error-prone task. The two semi-automatic ways also entail different costs: one exploits the available, static information and the other requires the system to be executed to compute a meaningful set of executions. We will investigate the trade-offs between modeling approaches and their diagnostic accuracy in the context of transparent computing.

[1]http://capec.mitre.org/

7 Conclusions

Identifying the root-cause and perform damage impact assessment of advanced persistent threats can be framed as a diagnostic problem. In this paper, we discuss an approach that leverages machine learning and model-based diagnosis techniques to reason about potential attacks.

Our approach classifies audit trails into high-level activities, and then reasons about those activities and their threat potential in real-time and forensically. By using the outcome of this reasoning to explain complex evi- dence of malicious behavior, the system administrators is provided with the proper tools to promptly react to, stop, and mitigate attacks.

References

[1] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2013.

[2] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. LogGC: Garbage collecting audit log. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, pages 1005–1016, Berlin, Germany, 2013.

[3] M. Galar, A. Fernández, E. Barrenechea, H. Bustince, and F. Herrera. A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches. *IEEE Transactions onSystems, Man, and Cybernetics, Part C: Applications and Reviews*, 42(4):463–484, July 2012.

[4] Charu C Aggarwal. Outlier ensembles: Position paper. *ACM SIGKDD Explorations Newsletter*, 14(2):49–58, 2013.

[5] Jing Gao and Pang-Ning Tan. Converting output scores from outlier detection algorithms into probability estimates. In *Proceedings of the Sixth International Conference on Data Mining*, pages 212–221. IEEE, December 2006.

[6] Hans-Peter Kriegel, Peer Kröger, Erich Schubert, and Arthur Zimek. Interpreting and unifying outlier scores. In *Proceedings of the Eleventh SIAM International Conference on Data Mining*, pages 13–24, April 2011.

[7] Erich Schubert, Remigius Wojdanowski, Arthur Zimek, and Hans-Peter Kriegel. On evaluation of outlier rankings and outlier scores. In *Proceedings of the Twelfth SIAM International Conference on Data Mining*, pages 1047–1058, April 2012.

[8] Hoda Eldardiry, Kumar Sricharan, Juan Liu, John Hanley, Bob Price, Oliver Brdiczka, and Eugene Bart. Multi-source fusion for anomaly detection: using across-domain and across-time peer-group consistency checks. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, 5(2):39–58, 2014.

[9] Yoav Freund, Raj D. Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *Journal of Machine Learning Research*, 4(Nov):933–969, 2003.

[10] Ke Deng, Simeng Han, Kate J Li, and Jun S Liu. Bayesian aggregation of order-based rank data. *Journal of the American Statistical Association*, 109(507):1023–1039, 2014.

[11] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the royal statistical society. Series B*, 39(1):1–38, 1977.

[12] Johan de Kleer, Olivier Raiman, and Mark Shirley. One step lookahead is pretty good. In *Readings in Model-Based Diagnosis*, pages 138–142. Morgan Kaufmann Publishers, San Francisco, CA, 1992.

[13] Alexander Feldman, Tolga Kurtoglu, Sriram Narasimhan, Scott Poll, David Garcia, Johan de Kleer, Lukas Kuhn, and Arjan van Gemund. Empirical evaluation of diagnostic algorithm performance using a generic framework. *International Journal of Prognostics and Health Management*, pages 1–28, 2010.

[14] Johan de Kleer and Brian Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[15] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M Wing. Automated generation and analysis of attack graphs. In *Proceeding of the 2002 IEEE Symposium on Security and Privacy*, pages 273–284. IEEE, May 2002.

[16] Seyit Ahmet Camtepe and Bülent Yener. Modeling and detection of complex attacks. In *Proceeding of the Third International Conference on Security and Privacy in Communications Networks*, pages 234–243, September 2007.

[17] Rui Abreu and Arjan JC van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *Proceedings of the Eighth Symposium on Abstraction, Reformulation and Approximation*, pages 2–9, July 2009.

[18] Alexander Feldman, Gregory Provan, and Arjan van Gemund. Approximate model-based diagnosis using greedy stochastic search. *Journal of Artificial Intelligence Research*, 38:371–413, 2010.

[19] Nuno Cardoso and Rui Abreu. A distributed approach to diagnosis candidate generation. In *Progress in Artificial Intelligence*, pages 175–186. Springer, 2013.

[20] Alexander Feldman, Jurryt Pietersma, and Arjan van Gemund. All roads lead to fault diagnosis: Model-based reasoning with LYDIA. In *Proceedings of the Eighteenth Belgium-Netherlands Conference on Artificial Intelligence (BNAIC'06), Namur, Belgium*, October 2006.

[21] Meera Sampath, Raja Sengupta, Stephane Lafortune, Kasim Sinnamohideen, and Demosthenis C Teneketzis. Failure diagnosis using discrete-event models. *Control Systems Technology, IEEE Transactions on*, 4(2):105–124, 1996.

[22] Alban Grastien, Marie-Odile Cordier, and Christine Largouët. Incremental diagnosis of discrete-event systems. In *DX*, 2005.

[23] Alban Grastien, Patrik Haslum, and Sylvie Thiébaux. Conflict-based diagnosis of discrete event systems: theory and practice. 2012.

[24] Behrooz Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, Inc., New York, NY, USA, 2nd edition, 2009.

[25] Janice Glasgow, Glenn Macewen, and Prakash Panangaden. A logic for reasoning about security. *ACM Transactions on Computer Systems*, 10(3):226–264, August 1992.

[26] Kenneth Forbus and Johan de Kleer. *Building Problem Solvers*. MIT Press, 1993.

[27] Johan de Kleer. Diagnosing multiple persistent and intermittent faults. In *Proceeding of the 2009 International Joint Conference on Artificial Intelligence*, pages 733–738, July 2009.

[28] Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. A new bayesian approach to multiple intermittent fault diagnosis. In *Proceeding of the 2009 International Joint Conference on Artificial Intelligence*, pages 653–658, July 2009.

[29] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99, November 2009.

[30] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing Verification and Reliability*, 10(3):171–194, 2000.