Attack Maze for Network Vulnerability Analysis

Computing the Maximum Possible Incursion and Intuitive Metrics

Stanley Chow
STHC Creative Technologies
Ottawa, Canada
e-mail: stanley.chow@pobox.com

Abstract— **Even a well administered computer network will be vulnerable to attacks. There have been many proposals in the literature to address the problem of Network-Vulnerability Analysis. One approach is to generate an attack graph (a logical graph representation of all possible sequences of vulnerabilities) using some formal model. Attack graphs suffer from scalability issues as the size of the network or the number of services and vulnerabilities increase. This paper presents a new approach that treats the network as a *maze*, which the attacker has to solve. We then use the classical way to solve mazes in computer games – remembering where we have been by dropping things at each node. We present a graph-based algorithm to solve this maze and compute the Maximum Possible Incursion (MPI) for a given set of attackers or compromises. The developed simple breadth-first algorithm provides performance improvements over previous approaches (less than a minute to analyze a network with over 10,000 nodes). We also present a methodology to capture mission dependency, which represents how a mission relies on the underlying network. Finally, we compute an extensible set of security metrics that identify the current network status in multiple dimensions (e.g. Confidentiality, Integrity, and Availability). We also discuss future work to enumerate the specific attack paths that could be used to generate corrective recommendations.**

Keywords- Network security; vulnerability analysis; scalable; vulnerability; exploit; maximum incursion; cyber security; metric; security metric; mission dependency.

I. INTRODUCTION

Cyber security has become more complex – the early generations of malware exploited a single vulnerability in a single computer system. Subsequently, worms and other malware propagate through a whole network. Recently, we have seen Stuxnet [1] and other sophisticated malware that use multiple vulnerabilities. Not only are malware getting more sophisticated, in many incidents, the attackers are known to have used a chain of vulnerabilities to gain access. There are many examples of such chains documented in various security advisories and so on.

Before we can analyze the possible chains of vulnerabilities, it is necessary to identify all the vulnerabilities present on each node. More generally, we need to identify the total attack surface of each node. Since there are many vulnerability scanners [2], and many agencies maintain databases of vulnerabilities, this paper assumes that all vulnerabilities are already known. It can also be difficult to capture the necessary network information, but this paper deals only with the analysis problem.

The problem of analyzing the many possible chains of vulnerabilities has attracted much attention. Most approaches ask: Can this node attack that node? One major approach is the *attack graph* introduced in 1998 [3]. Attack graphs are logical representations of all the ways an attacker could reach any target node in a given network. Although useful, attack graphs suffer from scalability in memory and performance issues as the network grows in number of nodes, services, vulnerabilities, etc. There are techniques in the literature that attempt to address the scalability of attack graphs in order to perform well for realistic-sized networks [4, 5]. This scalability problem is due to capturing all possible attack paths in the attack graph, so CPU time and memory usage grow rapidly with the size of the network. Another approach constructs an *access graph* of nodes in the network, where each directed edge in the graph represents a possible access along the edge [6].

We analyze the vulnerabilities for a different goal. Instead of calculating attack paths between specific nodes, we want to know exactly what privileges the attacker can possibly achieve – the *Maximum Possible Incursion* on each node. Clearly, this computation is specific to the particular class of attackers and must be recomputed for each class. Our approach, the *Attack Maze*, is similar to an access graph, but computes the MPI (Maximum Possible Incursion) directly. This means we do not record all possible Attack Paths, only the resultant incursion at each node – this is enough to achieve good scalability even for large networks.

Formal methods rely on accurately capturing all the intricacies of all the data – any missing data cannot be part of the inference chain. Some data are difficult to handle in formal systems, examples include: the privilege of a userid may be already in an LDAP (Lightweight Directory Access Protocol) directory and may change frequently – the difficulty is due to the unpredictable changes to the LDAP entry; the firewall may have rules that are dependent on time/data or even user – the difficulty is due to the sheer number of combinations that are possible and some dynamic rules that may include factors/variables not captured in the formal model, many transactions will depend on business logic (be it decision tree, decision tables, database look up or complex programmatic logic) - the difficulty is that many factors/variables may not be captured and that logic may be ill suited for the formal system. Since our approach is not based on a formal model, there is no need to precisely capture all details into the model; instead, the conditions can be embedded in code that is able to query LDAP, etc. (we do not allow arbitrary code - we require the code to respect monotonicity, see Step 5.d of the algorithm.)

The proposed approach also takes into account *mission dependency*. That is, given a mission that depends on some nodes of the network and given the current network status, what are the potential impacts on this mission? Some examples of mission dependency work in the cyber arena include [7, 8] and in the civil infrastructure area [9, 10].

We use the concept of *capabilities* to encapsulate what functions are exported by the network. Each mission can then use these capabilities without knowing the details of how they are implemented (e.g. which nodes provide email service).

We also present a suite of metrics that can be easily computed from the MPI. These metrics can be calculated at the levels of node, capability and mission, and have intuitive meaning to the owners of the node, capability or mission.

These ideas are implemented in a prototype using Python3 scripts. Our experiments show that even the simple algorithms perform very well – a well maintained network with few vulnerable nodes can be analyzed very quickly and even a network with many vulnerable nodes takes only minutes.

II. Attack Maze

A. Approach

Our approach is quite close to how an attacker tries to penetrate a network – find initial points of entry, then launch attacks from the compromised nodes to access more nodes and gain more privileges, repeat until no new privilege is possible. Along the way, the attacker keeps track of what access has already been achieved on each node, and only "better" accesses are of real interest. Eventually, all possible compromises on all nodes will be found. We define a *node* to be anything that is addressable (possibly with multiple addresses), so network printers, desktops, laptops, servers, proxies, are all nodes. We also generalize *firewalls* that control which nodes can access across *zone* boundaries.

B. Status

The key idea of the proposed algorithm is that we attach multiple *statuses* to each node. Each *status-type* records one particular type of privilege that the attacker can achieve at the node. The exact details of the statuses are expected to change with different applications (this paper presents some common statuses). Note that this algorithm does not rely on any specific status.

Each status-type should be at least a partial order – that is, the different levels of privilege should form a tree or hierarchy (as opposed to a complete order where the privilege forms a linear chain). We define *levels(s)* to be the number of levels in the hierarchy. The partial ordering of each status-type will induce a partial order on the whole node, that is, for nodes n_1 and n_2:

$n_1 > n_2$ iff $s(n_1) > s(n_2)$ for all status-types s

Note that there are two kinds of status-types:

- Status types that document increasing privilege,
  - None, anonymous shell, chroot jail, full user shell, root shell
  - None, write on /tmp only, write on ~/ only, write on anywhere
  - None, write file as anonymous, write file as user, write file as root
- Status types that document decreasing capability:
  - None (or Normal), 50% capacity, Non-functional (for example, measuring the capacity of a Domain Name Server)
  - Normal, some transaction over 100 millisecond, all transactions over 1 second (for example, measuring the throughput of a Web server)

C. Attack Step

We start by looking at the following attack step:

Node *A* uses exploit *E* to attack node *T*

We will refer to node *A* as the *attacker,* exploit *E* as the *exploit vector,* and node *T* as the *target* or the *victim* (a target is the intended victim of the attack, whereas a victim is after the attack succeeds). Each attack step will have *pre-conditions* and *post-conditions*. In this design, we explicitly limit pre-conditions to be dependent only on the combination {*A*, *E, T*} and the post-conditions are limited to status-fields of the victim. In other words, the pre-conditions for a particular vector *E* may be dependent on the statuses of *A*, and the statuses of *T*; whereas the post-conditions can only be statuses of *T*. Intuitively, when node *A* launches an attack, the attack may use all the privileges already gained at *A* as well as the privledges already gained at *T*. After the attack succeeds, the privilege gained **must** be at *T*. Note that no other nodes may be a part of the pre-conditions nor the post-conditions.

For example, we allow pre-conditions such as status-type "UserAccount" must be at least "user shell account" and status-type "UserpPiv" must be at least "can execute arbitrary program" – as long as the requirement is only on *A* or *T*. This is inherent in the definiton of status-type.

Most formal models do not restrict free variable like "user has FTP access on **some** server" (e.g., MulVAL [11] uses Datalog/Prolog logic rules so there is no problem with using another variable that will bind to another node). We explicitly disallow them in the pre-conditions, but allow them in the programatic code with some restrictions. As will be seen in Section E, this ensures the efficiency of the algorithm.

The restriction on pre-conditions does limit the kinds of attack steps that can be modeled; but we allow the programatic code to check for the same conditions – although this check must be consistent, repeatable and respects the monotonicity (a node can only increase its possible attacks when its statuses go up). This monotonicity ensures that we never have to backtrack. With this relaxation, we can easily handle attack vectors that require multiple intermediate nodes to cooperate. This means the resultant lost of expressive power is only nominal and the vast majority of real attacks can be modeled exactly and easily.

D. *Solving the Attack Maze*

To solve the maze, we start with the attacker(s) and try all possible victims (by recursively trying all possible attack steps on all possible targets). This ensures that we will traverse all possible attack paths from all attackers; along the way, we track only the maximum incursion at each victim. We use the naïve breadth-first algorithm described as follows:

Step 1. Start with just the nodes, initializing each node to have *None* (the lowest state) for each status-type. Intuitively, this is a sea of islands that any attacker has to hop to get anywhere, and the attacker starts with no access to anything.

Step 2. Initialize *newWorkList* to be the set of nodes that the attacker is assumed to have compromised - all their own machines (in their own domains) plus our machines that has been compromised.) This is an input to the Attack Maze computation. The statuses for the attacker(s) are set to the maximum privileges achieved. Intuitively, this represents the initial set of accesses that the attacker has.

Step 3. Check *newWorkList*, if it is empty, then we are done. If it is not-empty, copy *newWorkList* to *workList*, set *newWorkList* to empty.

Step 4. Removing an attacker Node *A* from *workList*. (If *workList* is empty, got to Step 3.) Intuitively, we will attempt to launch attacks from this node.

Step 5. Go through every node *T* in the system as a possible target from attacker *A*. (After running through every node, go to Step 4 for the next attacker.) Check if node *A* can attack node *T*:

a. Node *T* has a vulnerability *V*

b. The vulnerability *V* must have an exploit *E*

c. Node *A* can reach the address/port on node *T* needed to exploit *E*

d. Node *A* meets the pre-conditions of exploit *E* (note, this is the place for the non-local checks that must respect node monotonicity)

Step 6. If all the conditions (in Step 5)

a. are <u>not met</u>, this attack step is not possible. Go to Step 4 for the next target.

b. are <u>met</u>, then this attack step succeeds. The post-conditions of exploit *E* are merged into the statuses of node *T*. That is, we record the maximum of each status-type (since each status-type must be a partial order, there will be a maximum). If any status is increased as a result, add node *T* to *newWorkList*.

E. *Analysis of performance*

For analysis of performance, we will use:

- n – number of nodes
- $s - \sum_0^n levels(s_i)$
- v – number of actual vulnerabilities or exploits

Since each node can only be added to the *workList* with an increase in status, and since the statuses are monotonic, each node can only be on the *workList* s times. Each time a node is on the *workList*, the algorithm will examine all possible attacks from that node, so the total work will be $O(s*n*n*v)$ and since s and v are independent of the network, they can be subsumed into the coefficients, so the total work is $O(n^2)$. Note that this is for the algorithm, but we allow (in step Step 5.d) the pre-condition check to do arbitrary computation. In our prototype, we did not rely on this.

We make several observations on aspects that are often difficult:

- Exactness – within the accuracy of our status-fields (and extended pre-condition checks), we compute the exact MPI (Maximal Possible Incursion). This is true even if the pre-conditions are not completely formalized (i.e. embedded in code).
- Multiples paths getting to a node – we handle each possible step, but the effects of the steps are merged at the node. This means we compute the MPI without enumerating all possible paths, we only enumerate all possible steps.
- Cycles in attack paths – each complete cycle is handled; no extra processing is caused by multiple cycles. This is all implicit in the merging of status at nodes.

F. *Practical performance*

In the preceding analysis, the number of times a node can be put onto the WorkList is bound by s, the number of steps in the statuses. In practice, the loop (Step 4) iterates in lockstep with each link in an attack chain; that is, we start at the attacker(s) and follow all attack paths/chains simultaneously, one link per iteration. Therefore, the number of iterations is usually equal to the length of the longest attack path (counting in nodes, which is 1 more than the length in links). Even though this is only changing the constants and does not affect big-O, it does mean we can freely add more status-types without significantly affecting run-time.

Some optimizations that do not change the big-O, but can save significant time, are possible. For example, in step Step 5.c, instead of trying every node, we could try just the reachable nodes (either grouped by subnets or by nodes). It is also possible to precompute the attack surface of each node so that step Step 5.b becomes trivial. With these types of optimizations, the algorithm can hope to get close to O(n) on average, although O(n^2) is still the worst case. Note that, as we show later, n (the total number of nodes) is a poor predictor of performance; different types of node can have different impact – by factor of thousands.

G. *Prototype*

Our Python3 prototype is intended as a light weight, flexible, easy to use experimental test bed. The system is controlled by a control file (usually filetype ".maze") that controls every aspect of operation – the input data, the processing, the options, the output, the debugging. The control files processing (around 1K lines of Python) implements many facilities: nested includes, comments, timing, conditional jumps, setting of variables (such as the debug level), printing out data, sequencing operations.

We implemented a *Data Model* that includes *firewalls, zone, nodes, vulnerability,* etc. The Data Model is also around 1K lines of Python. The Attack Maze and the metrics total another 1K lines of Python.

The Attack Maze code has several parts:

- Status – code to handle definition of status-type
- Maze – algorithm to solve the maze
- Rules – the specific attack steps implemented as Python functions.

The rules are just individual attack steps. For example, this rule from MulVAL [11]:

```
accessFile(P, H, Access, Path) :-
   execCode(P, H, Owner), filePath(H, Owner, Path).
```

says "if an attacker P can access machine H with Owner's privilege, then he can have arbitrary access to files owned by Owner". Our equivalent Python code is show below in Figure 1:

```
have_priv=lookup_status(dfd_node.statuses, "Privilege")
if (have_priv > 1): ## have root priv
        ## is there any desired data on this machine?
        dfd_data = lookup_host_properties(dfd_node.host,
                                                  "Data_Bind")
        if (0 < len(dfd_data)):
              ## yes, so this succeeds
              updated=updateStatus(1,"GotData",
                                        dfd_node.statuses)
```

Figure 1. Python code example

In Figure 1, *dfd_node* is the target. We first lookup it's status of Privilege into *have_priv*, then check whether root privilege has been achieved. If it has, then, we lookup whether it has any *data binding* (MulVAL [11] term for data that the attacker wants). If both conditions are met, then the post-condition of *GotData* is set to record that this node will leak that data.

H. *Examples*

For our sample network , we start with the example from [11] and add the watering-hole attack from [12]. The network is the usual 2-firewall with DMZ. (DeMilitarized Zone.) Connectivity is shown in blue. For simplicity, each zone is assumed to be flat – any node can talk to any other node. The attack, which is from the Internet, takes 3 steps and is shown in red.

While running the algorithm, the *workList* will be: on iteration 1 {Attacker}, on iteration 2 {WebServer}, on iteration 3 {FileServer}, and, finally, on iteration 4 {Workstation}. So, a chain of 3 steps needs 4 iterations, as expected. We also include a node WorkSafe that is like WorkStation but without the vulnerabilities. In a well maintained network, most of the node will be of the WorkSafe variety (in a primitive way, the proportion of WorkSafe nodes serves as a measure of the security of the network.)

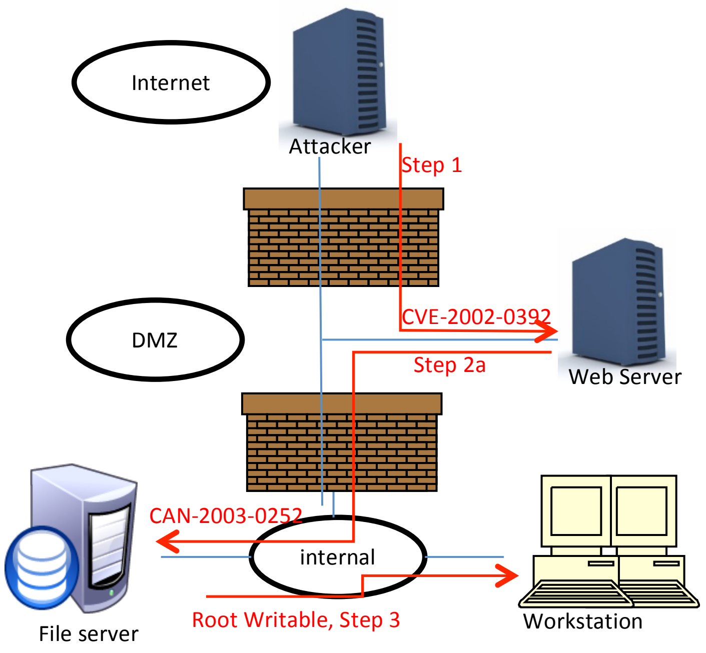

Figure 2. Test network

I. *Timings*

Our prototype implemented a "clone" directive to clone many copies of a node to test the scalability. Since we expect different behaviours for different types of nodes, we set up a number of scenarios listed in Table 1:

- Victim – vary the number of victims (cloning WorkStation up to 9K times)
- Innocent – vary the number of innocent bystanders (clone WorkSafe)
- Intermediate – vary the number of attack path intermediate nodes (clone WebServer)
- 3 X 1K – a fix 1K of each Victim/WorkStation, Innocent/WorkSafe, Intermediate/WebServer

Table 1. SCALABILITY CASES

Scenrio	WorkStation	WorkSafe	WebServer
Victim	1…9K	10	1
Innocent	10	1…9K	1
Intermediate	5	10	1…9K
3 X 1K	1K	1K	1K

The timings are done on an Dell XPS laptop with Intel i5-4210U CPU at 1.7GHz, 8GiB memory, Ubuntu 14.04 LTS, Python version 3.4.3a and times are reported in seconds of CPU time. Note that the memory usage for all cases was under 0.5 GiB and entirely in main memory, the script is single threaded, so multi-process is irrelevant. The data points are average of several runs.

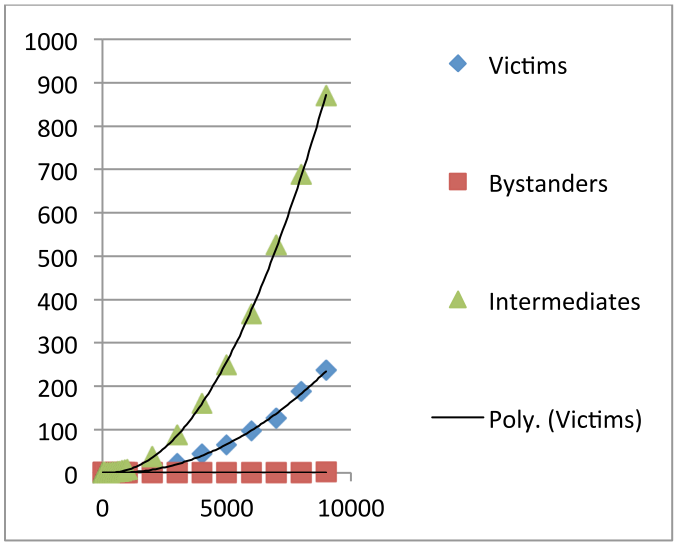

Figure 3. Scalability timing

In Figure 3, the green triangles are the number of vulnerable Web Servers (the intermediate stop in the attack path), the red squares are "safe" work stations (not in any attack path), and the blue diamonds are the WorkStations (victims). Not surprisingly, the timings all fit O(n^2) very well with R^2 values well above 0.99. On the other hand, the coefficients are quite different – 1e-5, 3e-6, 3e-9 respectively, or in ratio 3K:1K:1; this means "safe" nodes take practically no time, so a large well maintained network can be analysed in seconds. The victim nodes take more time, but even 10K victims take only a few minutes. The intermediate nodes are the most time consuming – 10K intermediates take around 20 minutes.

We also ran a case of 1K intermediates, 1K victims, 1K safes, for a total of 3K nodes (to be exact, we make that many clones of each type, but the network includes firewalls and other house keeping nodes, so the actual number of nodes is 3,026). It took around 40 seconds. This shows that even without any optimizations, it is entirely feasible for a network of realistic size.

III. Mission Dependency

To quote from [13] "It is critical that the [Department of Defense] develop better cognizance of Cyber Network Mission Dependencies". Some proposals, such as [14] are elaborate and somewhat difficult to construct. For example, a mission commander may know a particular mission needs email, but unlikely (may be even not allowed) to know which nodes are actually involving in provding email.

Our contribution is to define the concept of a *capability* which can be *exported* and *used*. The exporter is responsible to define how the capability is *implemented*, for example, in terms of nodes that are required. The *user* merely has to use the capability without knowing which nodes are involved.

This fits the real world situation quite nicely. For example, corporate IT may provide email, File Server, Print Server etc. while different groups may provide Sales Data, Inventory Data. A branch office IT can simply make use of these capabilities, and the system can resolve the dependencies. If the nodes that implement email are replaced or renamed, the users do not need to know (and probably will not know)!

This concept can extend to physical infrastructure like cables, buildings. It is also possible to capture redundancy requirements into the implementation of each capability. For example, the email capability requires just one of two nodes to be working (along with DNS capability).

IV. Metrics

There are different kinds of metrics for Situation Awareness: the patch status of each node, the attack surface of each node, where are the critical assets, active attacks in progress, etc., see [15] for a survey. Eigenvalues have been proposed as a mechanism for computing metrics, but they generally are not intuitive – a localized change can affect the metrics of nodes far away, for no clear reason. Even the sign of the change may be unpredictable.

We are interested in quantitative measures that are intuitive for questions like:

- How much damage can an attacker do? (For different classes of attackers)
- Which particular assets are vulnerable (to that class of attackers)?
- Is my particular mission safe – according to my requirements of the nodes and Confidentiality, Integrity, and Availability?
- Why did this metric go up? Because this particular attack path has been prevented by this particular patch.
- Assuming a new exploit, what will happen to the different missions?

To answer our kinds of questions, we start by solving the Attack Maze (for that class of attacker), so we know the MPI (Maximum Possible Incursion) at each node. Note that the status-types should be defined for the metrics. For example, in Figure 1. Python code example, the metric *GotData* records whether a node can access that particular data. Presumably, this particular fact is used in the metric calculation.

We then proceed to calculate metrics. We have several kinds of metrics:

- Self-metric – these metrics describe only what has happened to a node, ignoring other nodes.
- Other-metric – these metrics consider what this node can do to other nodes (give the MPI).
- Mission-metrics – these metrics are for missions, knowing the implementations of each capability, and the self-metric and other-metric of each underlying node.

A. *Metric Routine supplied by user*

We rely on the users to compute metrics from the MPI. That is, the user provides a routine to compute a metric for a node given the MPI. This allows users to link metrics to resources that are monitored:

- One group may have sales data that needs to be confidential, so they define a metric Sales_Confidential that is 0 or 1.
- Another group, say HR, may have salary data that also needs to be confidential, they define Salary_Confidential that is 0.0 to 1.0 depending on the difficulty of accessing that data (the evaluation routine will need attack models and other information that is not in the prototype, but there is no limit in principle).
- Another group may want a Web site to be available to the public, so they define Site_Available that is 0.0 to 1.0 depending on the state of DDoS (Distributed Denial of Service) attacks and how many servers are still up.
- A mission may define a metric Mission_Up from 0.0 to 1.0 to mean the percentage of capabilities and nodes are up. Of course, it does not need to be linear – the routine can set it arbitrarily.

B. *Self-metric*

Self-metrics are easily calculated – just invoke the associated user routine for each node. The meaning is explicitly narrow – the metric *Sale_Confidential* on a node means only whether attacker **on the node itself** can access the sales data.

The key is that the self-metrics form a "summary" for what a node can do, and we use self-metrics as the basis of mission-metrics.

C. *Mission-metric*

Mission-metrics are also computed by routines supplied by the user. These routines start with the self-metric for each node (that is needed for the mission), and produces metric for the mission. In our prototype, we favor the use of the "max" function. That is, the metric *Sale_Confidential* for the mission is just the max of the metric for each node. That is, the sales data is confidential in the mission if and only if it is confidential for each node.

D. *Example*

```
metric Confidentiality: max
        GotData,            No_Data=0.0, Got_Data=1.0
end metric Confidentiality

posture WorstC: max, "itemgetter('Confidentiality')"
```

Figure 4. Sample Metrics

This defines a metric *Confidentiality* that is 0.0 if the data is not compromised, or 1.0 if it is. Recall that this metric is computed for each node.

The posture (or mission-metric) *WorstC* is computed by taking each node, using Python itemgetter to get the *Confidentiality* metric, then take the *max* over all the nodes. In other words, this posture is indicative of whether a data leak is possible.

V. CONCLUSION AND FUTURE WORK

When solving the attack maze, it is relative simple to remember each (successful) attack step; that makes it possible to enumerate each possible attack path. The attack paths can be used to generate recommendations for securing the network. For example, it may be that there are many paths, but all the paths share a single link, in which case, patching a single machine may block all the paths. Essentially, we are trying to partition the network so that the attackers cannot get to the assets.

The attack maze can also be solved backward as well – that can tell us what privileges are required to get to a particular asset.

The capabilities concept can be expaned to deal with redundancy – n out of m, 75% capacity, etc.

ACKNOWLEDGMENT

This work was initiated by the author while on contract at Defence Research and Development Canada. We are grateful to Dr. Natalie Nakhla for helpful reviews and discussions.

REFERENCES

[1] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy,* vol. 9, no. 3, pp. 49-51, 2011.

[2] T. N. SecurityTM, "Nessus Open Source Vulnerability Scanner Project," ed, 2005.

[3] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," in *Proceedings of the 1998 workshop on New security paradigms*, 1998, pp. 71-79: ACM.

[4] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proceedings of the 9th ACM*

Conference on Computer and Communications Security, 2002, pp. 217-224: ACM.

[5] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM conference on Computer and communications security*, 2006, pp. 336-345: ACM.

[6] P. Ammann, J. Pamula, R. Ritchey, and J. d. Street, "A host-based approach to network attack chaining analysis," in *Computer Security Applications Conference, 21st Annual*, 2005, pp. 10 pp.-84: IEEE.

[7] P. A. Porras, M. W. Fong, and A. Valdes, "A mission-impact-based approach to INFOSEC alarm correlation," in *International Workshop on Recent Advances in Intrusion Detection*, 2002, pp. 95-114: Springer.

[8] G. Jakobson, "Mission cyber security situation assessment using impact dependency graphs," in *Information Fusion (FUSION), 2011 Proceedings of the 14th International Conference on*, 2011, pp. 1-8: IEEE.

[9] A. Antelman, J. J. Dempsey, and B. Brodt, "Mission dependency index-a metric for determining infrastructure criticality," *Infrastructure Reporting and Asset Management,* pp. 141-46, 2008.

[10] P. R. Garvey and C. A. Pinto, "Introduction to functional dependency network analysis," in *The MITRE Corporation and Old Dominion, Second International Symposium on Engineering Systems, MIT, Cambridge, Massachusetts*, 2009.

[11] X. Ou, S. Govindavajhala, and A. W. Appel, "MulVAL: A Logic-based Network Security Analyzer," in *USENIX security*, 2005.

[12] D. Kindlund, "Holyday watering hole attack proves difficult to detect and defend against," *ISSA J,* vol. 11, pp. 10-12, 2013.

[13] A. Schulz, M. Kotson, and J. Zipkin, "Cyber Network Mission Dependencies," ed: MIT Lincoln Laboratory, Tech. Rep, 2015.

[14] W. Heinbockel, S. Noel, and J. Curbo, "Mission Dependency Modeling for Cyber Situational Awareness." https://ist.gmu.edu/~csis/noel/pubs/2016_NATO_IST_148.pdf

[15] U. Franke and J. Brynielsson, "Cyber situational awareness–a systematic review of the literature," *Computers & Security,* vol. 46, pp. 18-31, 2014.