GUARD THE GAP

WHITEPAPER:
PATCH GAP ANALYSIS -
SECURING CRITICAL INFRASTRUCTURE

CHRIS THOMAS
FOXGUARD SOLUTIONS
2285 PROSPECT DRIVE
CHRISTIANSBURG, VA 24073
FOXGUARDSOLUTIONS.COM

BILL JOHNSON
TDi TECHNOLOGIES
1600 10TH STREET, SUITE B
PLANO, TX 75074
TDITECHNOLOGIES.COM

GUARD THE GAP

There is a moment when even the most seasoned IT professional's heart stops. After a new patch or a new update is installed on some critical piece of hardware he reaches out, flips the reset switch, and waits with bated breath. There is a moment of darkness and then, the lights come on, the disk spins up, and the spell is broken. Usually.

The truth is, not all updates work and some that fail can fail spectacularly. Every patch and every update carries with it a risk that the system and the millions of dollars in hardware and business that depend upon it will not come back once the lights go off. This is often why patches are mitigated rather than installed and why upgrades are often scheduled rather than applied as they become available. Even though the planned upgrade process may cost millions of dollars per day and delay installation by months or years, it is far preferable to the catastrophic failure of critical infrastructure at an inopportune moment.

Patch Gap Analysis is important because of the exorbitant costs, both of upgrading and failing to upgrade critical infrastructure. Understanding patch gap can drive down the need to mitigate and install updates, thereby reducing the risk of failure and the cost of security. Examining patches in the context of the vendor's structure can eliminate the need to install some of and highlight those most important to maintaining secure operation. Gap analysis can reduce or eliminate much of the attendant expense, downtime, and uncertainty of the patching process.

TRYPOPHOBIA

From the Greek trýpa meaning "drilling holes" and phobos meaning "fear," trypophobia is the fear of small holes or gaps and a common reaction to the problem of patch gap analysis. A vendor might conventionally present patches or updates to its product as a series (A, B, C, D, etc.) and will often list them as such alongside notes, dependencies, and other meta-data describing how those patches relate to each other. Considering these patches as presented in a sequential list would be overly simplistic and could lead to incorrect, even dangerous assumptions. Indeed, with just these four patches there are more than 11,000 distinct ways in which the patch vendor might have structured the network of relationships between them.[1] A given patch may include security information or it may be an expanded feature set; it may replace a previous patch or depend upon an earlier one. Patches may cause the product line to diverge, to converge, and even to do both simultaneously. In some of these more complex scenarios installing the patches in order of release, rather than in observance of the dependencies laid out by the vendor, could leave equipment more vulnerable and less functional than skipping the updates altogether.

Additionally complicating this problem is the issue of uneven patch application. As patching may not happen simultaneously across a given facility or company, the patches applied to one system may be replaced by one or more new releases before another system is able to be updated. In essence, the current state of the system being patched becomes a new variable in the complexity of any gap analysis, multiplying, rather than adding to, the complexity of the initial problem and resulting in hundreds of thousands if not millions of distinct combinations to consider on a device-by-device basis.[2]

Most organizations are grappling with the Gap Analysis problem whether they intend to or not. The actual patching process for any given piece of equipment will involve an applicability study in which a senior technician or some other trusted, responsible party will assess the patches and updates available and determines what to install based upon his understanding of the updates, the state of the system, vulnerabilities, and other factors. This intensely manual, expertise driven approach is where gaps are traditionally identified and mitigations drafted but it suffers from the drawbacks common to all manual processes: it is expensive to scale and prone to error. An automated approach is needed.

THE BOLLMAN TRUSS

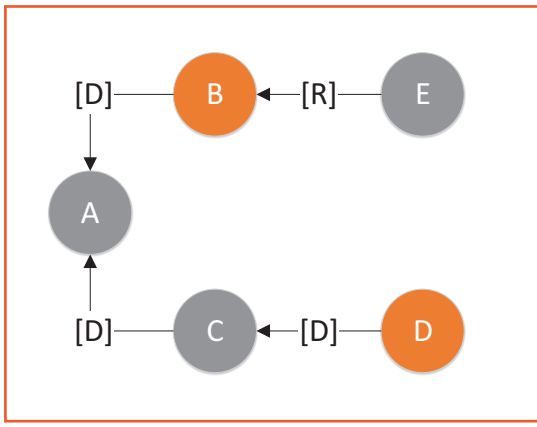
As the railroad spread across the American continent the need to bridge gaps cheaply, reliably, and safely became a key concern in the expansion of the rail network. The Bollman Truss is the only surviving example of a revolutionary design that made that expansion possible. Like the Bollman Truss, the solution to the patch gap problem lies in the web of connections and relationships which define the structure. Bollman used iron to expand the rail network but the key to solving the gap problem lies with social networks – specifically, with graph databases. The relationships between patches are a perfect analog to the relationships between the members of a set of friends and the technologies leveraged by social media giants like Facebook and LinkedIn are the ideal tools to traverse and identify gaps in the branching chains of patches, dependencies, and roll-ups common to information technology in critical infrastructure. By using a combination of graph and relational databases and then testing the output of those systems with orthogonal array testing methodologies it is possible to reliably, accurately, and efficiently compute a precise patch gap from among millions or even tens of millions of possible solutions.

GRAPH DATABASES

Graph databases are distinct from the relational databases more commonly used in software development. Relational databases store the relationships between data elements as data whereas these relationships are direct members of the graph data structure. As a result, graph databases allow the rapid traversal of massive, hierarchical structures in a single operation. Unfortunately, while relational databases have enjoyed several decades of mainstream use in the software development field, graph databases have only recently come into common usage – in large part due to the rise of social networks – and standards, toolsets, and expertise in them are scarce.

Graph databases are commonly used in social media applications because they allow rapid traversal of numerous, interconnected chains of relationships. When Facebook suggests new friends to its users it does so based not just on the friends they have in common (Alice and Bob sharing a common friend Carol) but based upon second and third degree relationships, interests shared across those relationships, and other pieces of data inherent to both the users and the relationships in question. For example, if Alice enjoys Star Wars and has checked in at The First Pastafarian Church she will likely be prompted to add Bob as a friend because Bob is friends with Charlie, the pastor of that church, and recently attended a screening of Rogue One with Carol, who is married to Bonnie, who is friends with Alice. The relationships here are "friends with" and "present at" and "interested in" but the ideas translate well to other domains.

Understanding these relationships and finding linkages between them relationally teeters on the brink of "impossible" but once projected onto a graph the relationships between people or between various patches and their constituent data elements are fairly simplistic. A patch may either replace another patch or depend upon it and it may either contain security information or not. These simple structures enable a complex traversal of the patch graph, creating useful emergent behaviors.



In the example above patches "B" and "C" depend upon patch "A" while patch "E" replaces patch "B" and patch "D" depends upon patch "C." Patches "B" and "D" are shown in orange, indicating that they contain security updates. This

GUARD THE GAP

THE BOLLMAN TRUSS (CONTINUED)

illustration thus shows five patches with four relationships between them with two properties per patch – a name and a security designation. Assuming a device at a patch level of "A," the patch gap here is "E, C, and D." Patch "B" need not be reported: it is replaced by "E." Patch "E," though not marked as containing security updates necessarily has security implications due to its replacement of "B" and should be reported as such. Patch "C," though not marked as a security update, is a dependency of patch "D" which has security updates and thus also has security value by association. These relationships change as the devices' patch level changes. If patch "B" were already present on the device patch "E" would lose its security status; whatever security benefits it might offer are already presumably conferred by the presence of patch "B."

As the graph grows in complexity so also do the possible permutations of output. In a relational structure, each layer of the graph would necessitate an additional query with additional joins. In a relational implementation, the size of the database as well as the time to query it grows exponentially with the addition of new data elements. In a graph, the database size and the time to query grow relative to log (elements).

ORTHOGONAL ARRAY TESTING STRATEGY (OATS)

To quote Agatha Christie, "a human error is nothing to what a computer can do if it tries." With just five patches, two relationship types, and two properties there are already a staggering number of combinations, structures, and outputs possible. Exhaustively testing them all to prove proper function of a solution is not only cost prohibitive but very likely impossible given the volume of real-world data. Yet testing is the only way to assure confidence in an automated solution.

To be clear, any testing short of exhaustive testing leaves an opening for error. Even a well architected solution with strict separation of concerns and careful management of internal state is prone to complex behavior based upon the complexities that arise as inputs interact with each other. That said, Orthogonal Array Testing provides a means to extract the maximum value and confidence from a given subset of exhaustive tests given that the inputs in question are orthogonal to each other – that is, given that they are logically distinct and statistically independent.

While the math is daunting, the core of OATS is the supposition that the vast majority of errors in a software application do not result from complex interactions but, rather, from pairwise interactions. In the words of Jeremy Harrell:

> Most of these defects are not a result of complex interactions such as "When the background is blue and the font is Arial and the layout has menus on the right and the images are large and it's a Thursday then the tables don't line up properly." Most of these defects arise from simple pair-wise interactions such as "When the font is Arial and the menus are on the right the tables don't line up properly."

Testing is then undertaken using an array of such pairwise variables ensuring that every possible pairwise combination is tested at least once. This dramatically cuts the number of tests necessary to establish confidence in the system. There still exists, of course, the possibility that a complex interaction (Arial, right menus, and large images on Thursday) will break the system but that possibility is unlikely and, if identified, can be rectified with suitable preventive tests to protect against future breakage.

SCANNING AND MINING

Of course the last, and probably most important, component of any digital solution is the ability to get data into it. For patch gap, that data amounts to patches, device patch levels, and the relationships between and among both.

Patches themselves are a data gathering and expertise issue. While the ISA-TR62443-2-3 specification provides a standard for representing patch data, it has not reached widespread acceptance and does not feature an accompanying standard for online patch discoverability, making the acquisition of patch data challenging at scale. Further, numerous IT and OT vendors restrict patching services to customers or sell such services as a feature of service agreements. In both cases this makes automated discovery, acquisition, and modeling of patch structures a difficult proposition.

While identifying installed patches does not suffer from the same problem of deliberate obscurity, it is nonetheless an equally challenging prospect. Calculating a patch gap means regularly assessing the patches already installed on a given device and comparing the results of that assessment against the patches available. Doing so requires not only the means to interrogate the device but the ability to determine from that interrogation which patches are installed. In many cases this comes down to case-by-case expertise and a fastidiously maintained, cross-linked dictionary of hard won key-value pairs.

Unlike the structural problems addressed above, the issue of repeated scanning and mining has no technological solution because no standard, repeatable algorithm – however inefficient – exists to extract and normalize the data. There are efficiencies to be gained and small pieces of the automation puzzle that an astute developer might fit together, but the problem as a whole is highly resistant to automation. Human effort, expertise, and adaptability are an inevitable and constraining component of this solution.

THE BREACH OF HARFLEUR

Charging headlong into this problem, like Henry V at the siege of Harfleur – "once more unto the breach dear friends" – is a recipe for disaster. While the tools and techniques outlined above are sufficient to solve the problem of patch gap analysis for a critical infrastructure provider, they pose a problem of their own. The size and complexity of the effort necessary to implement a solution, even with all of the tools identified and lessons learned already in hand is daunting to say the least. Even those with pre-existing software development teams would be forced to cross-train those teams on unfamiliar technologies, build, test, and deploy novel tools in unfamiliar environments, and staff teams to use those tools in a round-the-clock effort to populate them with high-quality data.

Such a Fordist approach is trading one expensive, difficult to maintain, and error prone problem for another. Accumulating the expertise, software, hardware, testing methodologies, and data necessary to develop an in-house patch-gap solution could very well obviate all of the financial incentives for having such a system in the first place. The solution is a vertical disintegration of the patch gap problem – buying, rather than developing the software development, data acquisition, and reporting knowledge necessary to find the gap. By outsourcing the identification and calculation of a patch gap the industries which provide and maintain critical infrastructure can spread the costs of gap analysis across many companies, driving down costs and improving both security and efficiency.

GUARD THE GAP

CREPUSCULAR RAYS

FoxGuard and TDi Technologies have partnered with the Department of Energy's Cybersecurity for Energy Delivery Systems program to illuminate the challenges of gap-based patching and to provide it as a service. The project initiative is to "simplify patch management for energy delivery industrial control systems". As a result, we have been focusing on simplifying many aspects of patch management, including, but not limited to, patch gap analysis.

The FoxGuard/TDi partnership puts the power of FoxGuard's proprietary patch gap engine and its IT/OT patch database – the largest in the world – behind TDi's ConsoleWorks agentless management solution to deliver per-asset, patch gap analysis at the push of a button. Users can see which systems are current, which need patches, what those patches are, and the security status of each of them. More importantly, they can be secure in the knowledge that patches reported are applicable to the system in question and not the start of another wild goose chase after another update that does nothing and threatens an expensive shutdown.

> "The hardest thing to learn in life is which bridge to cross and which to burn."
>
> – David Russell

[1] With four patches there are 4! (4x3x2x1 = 24) possible relationships between the various patches. Each of these relationships can be of at least three distinct types: replacement, dependency, or independence. This yields 243 = 13,824 possible relationship graphs. Accounting for metadata and circularity adjusts these figures somewhat and models may vary but 11,000 represents a conservative figure for relationship complexity here.

[2] Because different relationship types imply different results based upon the presence or absence of a given patch they may be mathematically modeled by doubling the number of relationships: Replaces-Absent, Replaces-Present, Dependency-Present, Dependency-Absent, etc. This means that the previous graph of 24 relationships with 3 relationship types becomes 24 relationships with 6 relationship types: 246 = 191,102,976. Again taking circularity and non-orthogonal metadata into account can reduce this complexity somewhat but the problem is conservatively one with millions of possible outcomes.

LEARN MORE ABOUT OUR PATCH MANAGEMENT SOLUTIONS

TDi Technologies and FoxGuard Solutions provide a wide range of patch management solutions that help companies identify and mitigate gaps in the security of their systems and prepare for NERC CIP audits.

For more information visit our company websites:
tditechnologies.com I foxguardsolutions.com

In addition, a weekly webinar is available to discuss ways to develop and implement a robust patch management program. To reserve a spot, please visit us at:

http://foxguardsolutions.com/patch-management-webinar

If you would like our security experts to contact you, then kindly share your contact details and a brief summary of your challenges at:

http://foxguardsolutions.com/contact-us/

FOR MORE INFORMATION:

To find out more about our services contact us at:

www.foxguardsolutions.com

requestinfo@foxguardsolutions.com

877.446.4732

company/717871

@FoxGuardInc

tdi technologies

FoxGuard Solutions®