Automatic Integrity Checks for Remote Web Resources

Existing tools for automatically detecting Web site defacement compare monitored Web resources with uncorrupted copies of the content kept in a safe place. This can be an expensive and difficult task, especially when working with dynamic resources. In contrast, the Goldrake tool uses sensors and alarms to automatically monitor remote Web resources' integrity. Such a framework could help designers build services to inexpensively monitor multiple Web sites, which could be very attractive for small, budget-limited organizations that depend on the Web for their operation.

Alberto Bartoli and Eric Medvet
University of Trieste

Web site defacement is one of the most common attacks on the Internet, with more than 490,000 Web sites defaced in the past year alone.[1] In this context, defacement is the fraudulent replacement of most or all of a site's content with other material aimed at making the intrusion evident to a visitor. Yet, an attacker that could perform a complete replacement could also perform subtler changes that would be difficult, if not impossible, for a user to spot — for example, modifying a script that collects password information and sends it back to the attacker. The relative ease by which an attacker can modify a Web site's content or appearance, coupled with the Web-based applications' ever-increasing penetration into daily life, underscores the urgent need to develop methodologies for effectively addressing this issue.

In our own research, we're studying how to detect this kind of attack automatically, immediately triggering a suitable reaction such as restoring the original version of the site or taking it offline for repair. All existing tools are based on some variation of the same basic idea (see the "Related Work in Web Defacement Detection" sidebar): the tool compares the defaced content with a trusted version of the site, kept someplace the attacker can't modify, and generates some form of alert if it detects changes. The site's administrators can be fully in charge of configuring and maintaining the tool, or an external entity can run the tool remotely, in which case the site's administrators have to provide and update the trusted version as appropriate.

In this article, we explore a different approach. We're interested in developing

 Published by the IEEE Computer Society 1089-7801/06/$20.00 © 2006 IEEE

Related Work in Web Defacement Detection

Most tools for detecting Web defacements are based on essentially the same idea: a copy of the site being monitored stays in a safe location for periodic comparisons. The tool generates an alert whenever a mismatch occurs. Such tools can run within the site being monitored[1] (such as WebAgain; www.lockstep.com) or remotely (such as Catbird; www.catbird.com). The site administrator often instructs the tool to skip content that's too dynamic for meaningful monitoring.

Tools that aren't devoted specifically to Web-defacement detection can still be used for that purpose. A file system integrity checker such as Tripwire, for example, detects changes in the parts of a file system that aren't supposed to change as well as deviations from a previously established baseline.[2] A more powerful and flexible approach is to use storage-based intrusion-detection systems, which also let the administrator specify an extensible set of update activities to be considered as suspicious.[3] Such tools must run within the site being monitored, though, and on carefully protected platforms.

Broadly speaking, our work is more similar to anomaly-based intrusion detection: our Goldrake tool observes a system to learn its behavior and raises an alert whenever it detects something unusual. The technical details also differ: rather than observing network traffic (like EMERALD)[4] or system call invocations,[5] Goldrake observes Web resources. Our work is fairly similar to Wenyin Liu and colleagues' SiteWatcher antiphishing tool.[6] In that case, the tool evaluates the visual similarity between a site observed across the Internet and a locally available sample of the original site.

References

1. S. Sedaghat, J. Pieprzyk, and E. Vossough, "On-the-Fly Web Content Integrity Check Boosts Users' Confidence," *Comm. ACM*, vol. 45, no. 11, 2002, pp. 33–37.
2. G. Kim and E. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker," *ACM Conf. Computer and Comm. Security*, ACM Press, 2004, pp. 18–29.
3. A. Pennington et al., "Storage-Based Intrusion Detection: Watching Storage Activity for Suspicious Behavior," *Proc. 12th Usenix Security Symp.*, Usenix Assoc., 2003, pp. 137–152.
4. P. Neumann and P. Porras, "Experience with EMERALD to Date," *Proc. 1st Usenix Workshop on Intrusion Detection and Network Monitoring*, Usenix Assoc., 1999, pp. 73–80.
5. D. Mutz et al., "Anomalous System Call Detection," *ACM Trans. Information and System Security*, vol. 9, no. 1, 2006, pp. 61–93.
6. W. Liu et al., "An Antiphishing Strategy Based on Visual Similarity Assessment," *IEEE Internet Computing*, vol. 10, no. 2, 2006, pp. 58–65.

a methodology for checking a remote Web resource's integrity automatically and without any participation from the monitored site — in particular, without requiring the installation of any infrastructure on that site. We want a software tool that can automatically learn the monitored resource's usual content and appearance, so it can generate alerts in case something unusual occurs. This task is particularly difficult because Web content is highly dynamic and Web resources can vary widely in structure and frequency of change. The challenge is how to develop a system that can keep false positives to a minimum while retaining the ability to generate meaningful alerts.

Goldrake

The main motivation for our work springs from the observation that, in our surroundings, there are many Web sites on which people depend, yet most of the organizations hosting them have neither the expertise nor the budget to appropriately preserve integrity. In our region alone, literally hundreds of Web sites represent various parts of civic administration, and it's easy to imagine the potential effects of malicious intrusions. However, very few of these organizations can afford to buy, install, configure, and maintain one of the existing tools or services based on the "trusted version" approach (indeed, the lack of adequate security budgets is a widespread problem today[2]). Our approach could change this scenario. A service capable of inexpensively checking remote Web sites' integrity — without placing additional burden on these sites' normal operation — could greatly improve Web-based computing infrastructures' robustness. We designed our Goldrake tool as a solution.

The Goldrake administrator specifies the set of Web resources the tool will monitor. For each resource in this set, Goldrake periodically retrieves the resource, analyzes its content, and executes a predefined task if the analysis suggests that an unauthorized change might have occurred. In practice, this last step could involve sending an alert to a specified paging device, but because such activity is outside of this article's scope, we won't cover it in detail here.

Goldrake's core is in step 2. First, the tool applies a set of *sensors* to a resource. These algorithms analyze a specific feature of the resource and return a Boolean value. A `true` return value indicates that the resource content is somewhat unusual or suspect. Then it sends the sensor's out-

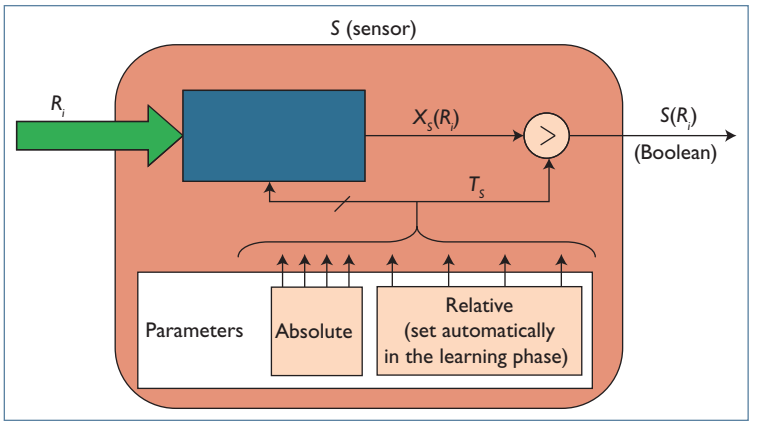

Figure 1. A Goldrake sensor's internal structure. Given a reading R_i, the sensor internally computes the indicator $X_S(R_i)$. The sensor then obtains the Boolean result $S(R_i)$ by comparing the indicator to the threshold to know if the indicator is greater than the threshold.

put as input to an *alarm*, which combines the results and returns another Boolean value. If the alarm still returns `true`, it indicates that the resource's integrity might have been compromised.

Obviously, a one-size-fits-all approach won't work in this context, so each resource should have its own set of sensors and alarms. Moreover, each sensor will have a set of parameters whose values can be tailored to the resource. Goldrake has a self-tuning ability: when given a new resource to monitor, the tool observes the resource for a specified learning period and then automatically chooses suitable parameter values.

We took great care to make our tool extensible. Goldrake administrators can add new sensors and alarms simply by developing classes that implement a particular interface (we wrote Goldrake in Java). This feature is crucial for both facilitating experimentation and increasing the attack space's coverage.

Sensors

Let's say a sensor S receives a sequence of readings for resource R as R_1, R_2, and so on, and provides as output the corresponding sequence of Boolean values $S(R_1)$, $S(R_2)$, and so on.

Each sensor is internally structured as in Figure 1. Goldrake inputs the current reading R_i to a parametric algorithm that produces a numerical index called the *indicator*, denoted as $X_S(R_i)$. If R is a dynamic resource, repeated readings of it could lead to different values for the indicator. The sensor then obtains the output by comparing the indicator to a numerical parameter T_S (the *threshold*): $S(R_i)$ is true if and only if $X_S(R_i) > T_S$.

In other words, the indicator can vary over time because R is dynamic. When the indicator falls outside its typical range, Goldrake treats the reading as suspicious — as a hint that an unauthorized change might have occurred. Of course, this heuristic approach doesn't provide a formal guarantee of correctness. As with any other anomaly-based heuristics, it could exhibit false positives (for instance, the sensor "fires," or raises an alarm, in response to legitimate changes to R) or false negatives (for example, the sensor fails to fire in response to illegitimate changes).

A sensor's parameters can be either relative or absolute. A relative parameter's value must be tailored to a specific resource. One of our sensors uses the average number of lines — say, η — for example, and the standard deviation σ. The value of indicator $X(R_i)$ in this case will be $|(n_i - \eta)|\sigma$, where n_i is the number of lines in the reading R_i. Clearly, η and σ are relative parameters, so Goldrake automatically determines their values in the learning phase. We can set an absolute parameter's value arbitrarily. A sensor that looks for a certain class of unwanted words, for example, would have a set of those words as an absolute parameter.

During the preliminary learning phase for a resource R, Goldrake constructs an archive of readings and then delegates to each sensor the job of finding suitable values for the respective relative parameters. We omit the full details here for the sake of brevity, but this procedure doesn't blindly trust all readings because some of them could include unauthorized changes.

Table 1 summarizes the 18 sensors we've implemented in our prototype. We group the sensors in four categories, as follows (L_R indicates the sequence of readings of resource R collected during the learning phase):

- *Cardinality* sensors count the number of items in a reading but differ in what an "item" really is. Let η_R and σ_R denote, respectively, the mean and standard deviation of item counts in L_R. The indicator for a given reading R_i is $|n_i - \eta_R)/\sigma_R|$, where n_i is that reading's item count.
- *Recurrent* sensors look for missing recurrent items but again differ in what an "item" really is. (An item is recurrent if it's present at least once in each reading of L_R.) Let $REC(L_R)$ be the set containing all recurrent items in L_R. The indicator for a given reading R_i is the ratio between the number of elements of $REC(L_R)$

Table 1. Sensors implemented in our current Goldrake prototype.

Cardinality	
TooManyOrTooFewBytes	Counts the number of bytes (size)
TooManyOrTooFewImages	Counts the number of images
TooManyOrTooFewLines	Counts the number of lines
TooManyOrTooFewStructuralTags	Counts the number of structural tags
Recurrent	
TooFewRecurrentAbsoluteLinks	Searches for missing recurrent absolute links
TooFewRecurrentExternalLinks	Searches for missing recurrent external links
TooFewRecurrentImages	Searches for missing recurrent images; two images are considered equal when they have the same name and digest
TooFewRecurrentNonNumericalAbsoluteLinks	Searches for missing recurrent nonnumerical absolute links (such links don't contain numeric characters in the URL)
TooFewRecurrentNonNumericalExternalLinks	Searches for missing recurrent nonnumerical external links
TooFewRecurrentNonNumericalImages	Searches for missing recurrent nonnumerical images
TooFewRecurrentNonNumericalLinks	Searches for missing recurrent nonnumerical links
TooFewRecurrentScripts	Searches for missing recurrent scripts; only considers inline scripts
TooFewRecurrentTags	Searches for missing recurrent tags; only considers composite tags (tags with a beginning and an ending tag)
TooFewRecurrentWords	Searches for missing recurrent words; only considers words contained in visible text blocks
Tree	
TooManyMissingNodesFromRecurrentElementTree	Tree nodes contain only HTML elements
TooManyMissingNodesFromRecurrentStructuralTagTree	Tree nodes contain only HTML structural tags, corresponding to HTML, BODY, HEAD, DIV, TABLE, TR, TD, FORM, FRAME, INPUT, TEXTAREA, STYLE, or SCRIPT
Frequency	
UnusualCharsFrequency	Computes relative frequencies of text characters; only considers those contained in visible text blocks
UnusualHtmlElementsFrequency	Computes relative frequencies of HTML elements

absent from R_i and the cardinality of $REC(L_R)$. Thus, if all elements of $REC(L_R)$ are in R_i, the indicator will be 0, whereas if no element of $REC(L_R)$ is in R_i, the indicator will be 1.

- *Tree* sensors look for missing portions of a resource's typical tree. Each reading is represented as a tree, which the sensor obtains from a specific transformation of the corresponding HTML/XML tree. Let $MCT(L_R)$ denote the maximum common tree among all the trees obtained from L_R. The sensor obtains the indicator for a given reading R_i in three steps: R_i is represented as a tree $T(R_i)$; the sensor constructs the maximum common tree $MCT(R_i)$ between $MCT(L_R)$ and $T(R_i)$; and the indicator is $1 - |MCT(R_i)|/|MCT(L_R)|$, where $|MCT(R_i)|$ and $|MCT(L_R)|$ are the number of nodes of $MCT(R_i)$ and $MCT(L_R)$, respectively. Thus, if $T(R_i)$ coincides with or is a supertree of $MCT(L_R)$, the indicator will be 0; if $T(R_i)$ and $MCT(L_R)$ are disjoint, the indicator will be 1.
- *Frequency* sensors measure items' relative frequencies. Let $R_F(L_R)$ denote the relative frequency of items resulting from L_R. The indicator for a given reading R_i is the sum of the relative differences (absolute values) between frequencies observed in R_i and those in $R_F(L_R)$.

For recurrent and tree sensors, we set the threshold to 0 — that is, a recurrent sensor fires whenever at least one recurrent item is missing. For cardinality and frequency sensors, we set the threshold based on observations in L_R. A detailed description of a sensor in each category appears elsewhere.[3] The sensor will trigger an alarm based on comparison between the indicator and the threshold.

What about attacks for which none of our sensors would fire — say, if an attacker gained control of a page to obtain customers' private data and added a script to that page? Our sensors might not detect such a modification (although TooFew RecurrentScripts would detect modifications). Nevertheless, a new sensor tailored to this kind of

Table 2. Alarms in Goldrake.

Name	Parameters and description
TooManyFiringSensors	Set of sensors, threshold; this alarm counts the number of sensors that return `true` (if this number reaches the threshold, the alarm evaluates to `true`)
TooManyFiringGroups	Set of sensors divided into subsets (groups), threshold; this alarm considers the output of sensors in the same group (if the number of groups that fire reaches the threshold, it evaluates to `true`)
SingleSensor	One sensor; this alarm simply reports the sensor's output (useful mostly for experimentation)

attack is easy to develop — a sort of TooManyOrTooFewScripts.

Although our approach doesn't require any knowledge about the monitored site, it could exploit such knowledge when available. For many sites, external links aren't supposed to contain sex-related words, for example, but to ensure that this is indeed the case, we could add a sensor that fired when one or more undesired words appeared in an external link.

Unfortunately, attacks that perform extremely localized changes — for example, "today is a sunny day" becomes "today is a rainy day" — are simply undetectable with our approach. This limitation is the obvious consequence of not relying on a trusted duplicated resource and is the price to be paid for remaining fully decoupled from the monitored site.

Alarms

An alarm also has a set of parameters that the tool administrator must set before use. Table 2 shows the three alarms in Goldrake.

The administrator defines the association between resources, alarms, and sensors. When an alarm evaluates to `true`, Goldrake executes a specified action, which, in our current prototype, is a shell command. It's possible to associate multiple alarms with a given resource, which can be useful for triggering different actions depending on the kind or severity of change detected.

Experiments

To investigate our tool's effectiveness, we performed several experiments on 11 total pages across six sites. We chose them to represent a suitable mix of real-world sites with both static and dynamic content:

- *Università degli Studi di Trieste* (www.units.it, the University of Trieste). For this site, we monitored only the homepage, which is mostly static but includes a small dynamic section for news.
- *Repubblica* (www.repubblica.it, a daily Italian newspaper). We monitored the homepage and the technology and science main page, which both change at least daily.
- *ANSA* (www.ansa.it, an Italian news agency). We monitored the homepage, whose content is highly dynamic, and the sport section's RSS feed.
- *Autoroutes de Sud de la France*, (www.asf.fr, the Southern France's highway company). We monitored the homepage and the traffic page.
- *Amazon.com*. We monitored only the homepage.
- *CNN.com*. We monitored the homepage, the main business page, and the main weather page, all of which have highly dynamic content.

For each page, we constructed a learning archive composed of 60 readings, fetched every six hours for 15 days. We performed all tests after Goldrake's learning phase.

Our first suite of tests counted the number of false positives. This kind of analysis is important for evaluating our approach's ability to cope with dynamic content while discriminating between legitimate and unauthorized changes. We monitored all pages for 20 days, fetching readings every six hours, totaling 80 evaluations for each <resource, sensor> pair. An operator's visual inspection of the resources showed that no unauthorized changes appeared to have occurred during our monitoring period.

In the next phase, we counted the number of false negatives. To this end, we simulated 20 different defacements for each page. In our attempt to mimic real-world defacements, we randomly extracted 20 defacements from the digital attacks archive at www.zone-h.org.

To gain initial insight into our approach's effectiveness, we first evaluated each sensor's behavior in isolation — that is, with the SingleSensor alarm. The full results of this preliminary analysis appear elsewhere,[3] but we found that tree-type sensors gave the best results. In particular, they showed no false negatives and only four false positives, all of

Table 3. Policies used in our experiments.

Policy name	Description
TooManyFiringSensors(1)	The alarm TooManyFiringSensors is applied to all sensors with threshold 1 (at least one sensor must fire)
TooManyFiringSensors(3)	Same as above, with threshold 3
TooManyFiringSensors(5)	Same as above, with threshold 5
TooManyFiringGroups(3)	TooManyFiringGroups, with all sensors, grouped as in Table 1, threshold 3
TooManyFiringGroups(4)	TooManyFiringGroups, with all sensors grouped as in Table 1, threshold 4 (all groups must fire)
AtLeastOneOfTreeSensors	TooManyFiringSensors with only the two tree sensors, threshold 1

which were concentrated on the same resource — the ANSA homepage — and exhibited by both sensors on the same four readings.

Because tree-type sensors offered ideal performance for all the pages we analyzed, you might wonder whether there's really any need for more complex alarms or other sensors. Unfortunately, we can't foresee all the detailed actions an attacker will perform, so it's impossible to prove from a benchmark that a detection system of this kind will never exhibit false negatives in practice. This is true for intrusion detection systems as well. Increasing the number of sensors is certainly useful, provided they focus on different features of the resource or analyze the same feature from different viewpoints. Quantifying the beneficial effects that could result from adding a certain set of sensors, however, is difficult to do a priori.

Following our preliminary analysis, we defined six alarms (see Table 3; we use the term *policy* to indicate an alarm with parameters) and applied them to all the pages. Figure 2 summarizes the results.

Our tests showed that all policies were basically able to detect simulated defacements in our setting, with false negatives reported only for TooManyFiringGroups(4) on three of the pages. Moreover, two policies showed no false positives at all, whereas two other policies exhibited a few false positives concentrated on one or two pages. These encouraging results tell us that we can effectively apply our approach in practice and that Goldrake administrators could indeed cope with dynamic content without being overwhelmed by a flood of false positives.

A more detailed analysis of the policies reveals several interesting findings. TooManyFiringSensors policies are the simplest and fire when at least one, three, or five sensors fire. Looking at the results in detail, an administrator could keep the number of false positives sufficiently low by increasing the minimum number of firing sensors that trigger the alarm. In fact, if the administrator requires that at least five sensors fire, the number of false positives drops to zero for most pages. However, making this adjustment might also increase the potential for unauthorized changes to go undetected. For the ASF France homepage, this approach simply isn't effective: the number of false positives remains excessively high even under the TooManyFiringSensors(5) policy.

TooManyFiringGroups policies are slightly more complex and much more effective. They don't consider sensors to be independent of each other, so they merge their results based on the categorizations in Table 1. The rationale is that sensor outputs in the same group tend to be correlated because sensors in a given group apply a similar algorithm on different features.

The next phase of our research is how to approach the detection problem when we consider longer monitoring sessions (months instead of weeks, as we've reported here). Our short-term goal is thus to study our tool's effectiveness in such scenarios. □

Acknowledgments

Michele Susel implemented an early version of Goldrake and contributed many useful ideas to this activity. Fulvio Sbroiavacca provided the initial motivation for this work. Fulvio Gini helped us better understand the basics of detection theory. The anonymous referees provided constructive comments. We gratefully acknowledge all this support.

References

1. "Statistics on Web Server Attacks for 2005," *The Internet Thermometer*, June 2006; www.zone-h.org/component/option,com_repository/Itemid,47/func,fileinfo/id,7771/.
2. T. Kemp, "Security's Shaky State," *Information Week*, 5 Dec. 2005; www.informationweek.com/story/showArticle.jhtml?articleID=174900279.
3. A. Bartoli and E. Medvet, *Automatic Integrity Checks for Remote Web Resources* (extended version), tech. report,

	False negatives (simulated defacements)						False positives (normal observation)					
	TooManyFiringGroups(4)	TooManyFiringGroups(3)	AtLeastOneOfTreeSensors	TooManyFiringSensors(1)	TooManyFiringSensors(3)	TooManyFiringSensors(5)	TooManyFiringGroups(4)	TooManyFiringGroups(3)	AtLeastOneOfTreeSensors	TooManyFiringSensors(1)	TooManyFiringSensors(3)	TooManyFiringSensors(5)
ASF France: Homepage	0	0	0	0	0	0	0	0	0	73	73	3
ASF France: Traffic	1	0	0	0	0	0	0	0	0	0	0	0
Amazon: Homepage	0	0	0	0	0	0	0	0	0	15	8	0
Ansa: Homepage	0	0	0	0	0	0	0	0	4	50	4	0
Ansa: RSS sport	1	0	0	0	0	0	0	0	0	0	0	0
CNN: Business	0	0	0	0	0	0	0	0	0	67	66	3
CNN: Home page	0	0	0	0	0	0	0	0	0	0	0	0
CNN: Weather	0	0	0	0	0	0	0	0	0	6	0	0
Repubblica: Homepage	0	0	0	0	0	0	0	0	0	58	22	0
Repubblica: Tecnologia e Scienze	0	0	0	0	0	0	0	0	0	50	50	0
Università Trieste: Homepage	1	0	0	0	0	0	0	0	0	30	15	0

Figure 2. False positives and false negatives for each <resource, policy> pair. We simulated 20 defacements to test Goldrake's ability to avoid false negatives and took 80 readings to test its ability to avoid false positives.

Univ. of Trieste, Jan. 2006; www.univ.trieste.it/bartolia/download/AutomaticIntegrityChecks.pdf.

Alberto Bartoli is an associate professor of computer engineering at the University of Trieste, Italy. His research interests focus on reliability in distributed systems. Bartoli has a BS in electronic engineering and a PhD in computer engineering, both from the University of Pisa, Italy. He is a member of the ACM. Contact him at bartolia@univ.trieste.it.

Eric Medvet is a PhD student in the computer engineering department at the University of Trieste, Italy, where he also earned a BS in electronic engineering. His research interests are in computer network security. Contact him at emedvet@units.it.