Static Analysis Alert Audits: Lexicon & Rules

David Svoboda, Lori Flynn, and Will Snavely
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania USA
Email: {svoboda, lflynn, wsnavely} @cert.org

***Abstract*—There is no widely-accepted lexicon or standard set of rules for auditing static analysis alerts in the software engineering community. Auditing rules and a lexicon should guide different auditors to make the same determination for an alert. Standard terms and processes are necessary so that initial determinations are correctly interpreted, which helps organizations reduce code flaws. They are also needed to improve the quality of audit data to benefit research on alert prioritization. This paper provides a suggested set of auditing rules and a lexicon, detailing rationales based on modern software engineering practices for each rule and each lexicon term. Some code examples are provided with the auditing rules. The authors' hope is that this suggested framework will motivate community discussion leading to agreed-upon standards.**

I. Introduction

In the software engineering community, there is no widely-accepted lexicon or standard set of rules for auditing static analysis (SA) tool alerts. This is in spite of static analysis being an integral part of the software development lifecycle [1] [2]. A standard auditing rule set and lexicon should guide different auditors to reach the same determination for an alert, achieving consistent results by reducing ambiguity. Well-defined auditing terms and processes would help organizations to reduce flaws during code repair, code development, and future audits. A standard lexicon and rules would also improve audit data quality, benefitting research on alert prioritization and classification (detailed in Section II). This paper starts with a general description of modern alert auditing and associated issues that shape a lexicon and auditing rules, noting related work (research as well as practice) on lexicons and auditing rules. Section II provides a suggested lexicon with a rationale for each term. Section III lists suggested auditing rules with a rationale for each rule. Section IV discusses a small test case. Section V summarizes conclusions and future work plans.

A. Static analysis

SA tools attempt to automatically identify defects in software products. At a high level, these tools define a set of *conditions* describing a well-behaved program (e.g., a null pointer shall not be dereferenced). SA tools then analyze a program to find violations of those conditions. The analysis may inspect the source code of the program, or some other representation thereof, such as the compiled binary.

However, SA tools often wrongly accuse programs of violating a condition. That is, static analysis is prone to false positives. Formally determining statically whether a program violates arbitrary conditions is undecidable. Hence, tools apply heuristics to find potential bugs, and these heuristics tend to over-approximate the occurrence of violations. False positives may also result from bugs in the tool.

Consequently, SA reports must be validated. This task often falls to human auditors.

B. State of the art: auditing

Auditors examine tool alerts and determine whether the program violates the condition identified therein. This determination may be a simple True or False (the code violates the condition, or it does not), though other labels are possible.

An audit determination alone does not indicate whether the offending code should be modified; such decisions are driven by organizational priorities. For example, an alert may be False but the related code still warrants fixing, e.g., if the code construct is difficult to maintain. Conversely, an alert may be True but not warrant fixing, e.g., because the code is dead or a fix elsewhere mitigates the problem. An organization may let management determine alert prioritization, or may set policies for that. For example, a zero-new-defect policy would mandate that new true alerts be fixed before new code is written.

Organizations can vary widely in their auditing sophistication. At one end, each developer performs their own SA, audits the results, and fixes bugs on an individual basis. At the other end, dedicated auditors do nothing but diagnose alerts without any alert prioritization. These auditors may be oblivious to audit resolution issues, and need only conduct audits based on the code. While audit resolution is an interesting problem, this paper ignores that problem and instead focuses on the auditors' problem of audit determinations.

For most large codebases, there are too many alerts for a team to economically address them all. Auditors may therefore triage alerts, preferring alerts in a certain category or with a certain (e.g., tool-reported) priority [3]. Some tools allow auditors to specify which alerts were not audited and why.

Software grows and evolves as bugs are fixed and features added. Some changes introduce new bugs. Consequently SA tools must be periodically re-run on the codebase. Some SA tools track a codebase's audit history, remembering previous verdicts. This is helpful if a True alert remains unfixed, e.g., if the fix is postponed. It is most helpful with false positives. Without this tracking ability, an auditor would have to mark the same alerts as false many times during development.

Automatic alert classification is an active area of research. Classification models have achieved high prediction accuracy in some studies [4].

C. Related research & existing SA tool functionality

Previous research publications use varied lexicons for static analysis determinations. True and False determinations (or synonyms) are used by all the research, and are often the only determinations mentioned (e.g., in Livshits' work on finding security vulnerabilities in Java applications [5]). Delaitre's rating system evolved multiple times during a research project using test suites to analyze static analysis tool efficacy, and the lexicon for alert determinations ended up being "security-related", "quality-related", "insignificant", and "false" [6]. Ciriello's analysis simply uses "C++ Test Problem" and "False Positive" determinations [7]. Baca's work on improving software security using static analysis in an industry setting uses three auditing determinations: "False positive", "True positive" for correctly identified faults that do not affect specified parameters with respect to security, and "Security" for a correct warning that could propagate into a user-induced system failure [1]. Carlsson's research with SA tools used four audit determinations: "false positive", "possible security improvement", "security risk with consequences", and "false negative" (the latter was possible because he used multiple SA tools) [8]. Cifuentes describes Oracle's internal deployment of the Parfait static analysis tool, where their server keeps track of historical audit data per codebase including nightly run results, overall report status, and audit determinations using a lexicon of "true", "false positive", and "won't fix" [9].

Much static analysis research disregards utility of historical determinations on alerts, for instance, if the focus is on utility of a particular new analysis technique to find a flaw. Johnson's research on why software developers don't use SA tools to find bugs discovered barriers including difficulty integrating SA tools into their development environment and development workflow [10]. An auditing determination lexicon with terms like "Dangerous construct", "Dead", "Complex", and "Current environment not applicable" (see Section II) could address those issues and be helpful in a development environment and workflow where previously dead code can be used, too-complex alerts can be avoided if work effort isn't available, and platform environments can change. Likewise, Chess' "Secure Programming with Static Analysis" book says that SA tool output needs to integrate easily with the development environment, with historical results being stored for future code reviews [11].

Most SA tools natively provide various alert determinations (differing per tool), and some provide an annotation capability.

Related work providing a set of documented auditing rules could not be found. General guidelines are provided in Chess' book, which recommends considering mitigation factors that could prevent the code from being vulnerable despite the SA tool's alert, and also recommends that the user pay attention to other problems they find inspecting the code even if there are no alerts for them (related to our rule 8).

D. Related practices by project collaborators and students

Our three DoD collaborator organizations had varied methods for auditing and for making audit determinations. None previously had a documented standard for rules or lexicons. One large organization used notably different auditing determinations in its different groups. For example, in one group "True" simply meant validation of the indicated error, but in others "True" was used to indicate code which should be fixed (e.g., the alert was false, but the code had a difficult-to-maintain construct). Most used the determination options provided by the SA tools they used. One auditor mentioned that code lines with associated historical "do not fix" determinations sometimes currently needed fixes, so more information should be noted (e.g., free form notes and/or a determination "not a problem in current environment"). Another auditor said his perception is that "we are more concerned with finding problems in our code than in minimizing variance between reviewers." Those goals are related since declaring "no problem with this code" could result in a missed defect.

Prior to this project, CERT also lacked documented auditing rules and a lexicon. In the past year, we have trained 3 classes of software engineers (2 were DoD organizations, and 1 was a class at Carnegie Mellon University – a total of 37 students) on these auditing rules and the lexicon, inviting feedback each time. None of the students reported having used an auditing lexicon or rules before, and their suggestions were incorporated into the lexicon and rules.

[12], [3], [13], [14], and [4] aim to determine if alerts are true, using automated and accurate methods, and require a large amount of high-quality archived audit data. That archived data is statistically analyzed to develop classifiers. High-quality audit determinations mean that different auditors should come to the same determination for the same SA alert. Aggregation of such audit archives would become possible and useful. Improving the auditing, code maintenance, and cross-program development within an organization and developing accurate classifiers and alert prioritization schemes in general will require consistent auditing methods and a standard lexicon.

II. Auditing Lexicon

The lack of a standard auditing lexicon (see Section I-D) can cause problems in code maintenance and cross-program development. For instance, a manual audit determination of "True" for one alert should mean the same thing to different organizations. However, for some organizations it is used even in situations where the alert is false, to signify "must fix" (e.g., if the code construct related to the alert is dangerous). Similarly, organizations deal with alerts related to dead code using determinations in overlapping and different ways.

In this section, we present a lexicon that we believe is necessary for common audit determinations. The lexicon is based on a survey of previous literature, our research with audit archives, our work analyzing corner cases for audit rules in Section III, our experience auditing 16 million LOC, and

consultation with three (anonymous) collaborating DoD organizations that together audit over 200 million LOC annually.

Manual audit determinations vary if the same process is not used. In developing our set of auditing rules, we found corner cases that we believe auditing systems should explicitly dictate how the auditor should handle those cases. For example, if an alert about one flaw depends on a second code flaw (e.g., one that could cause indeterminate behavior) to be True, should the first alert be marked True? Ideally, different auditors should reach the same determination for the same alert. Unfortunately, without more expressive determinations, different auditors can mark such cases differently. Some organizations mark as True any alert that warrants fixing, even if it is incorrect, while others mark such alerts as False.

Our audit determinations lexicon consists of labels we believe every auditing framework should provide, based on our expertise, related research review, review of modern tools, and consultation with professional code auditors including our project collaborators.

Our goal in developing the lexicon is to provide audit determination labels that will help efficiently direct work on the codebase, according to an organization's priorities and workers' varying time availability. "Workers" here means auditors and developers who repair the code after an audit. Those developers use the audit determinations to look for particular types of code flaws that their organization prioritizes fixing. We aimed to include all the labels (and combinations of labels) that the developers might want to target or filter out. Auditors may make initial determinations, and then later (only if time allows) complete work on alerts that previous auditors hadn't been able to resolve as True or False. Furthermore, archived audit determinations for previous versions of the codebase might be carried over by an auditing tool, to help guide auditors' work. Our lexicon is meant to be useful for auditing tools with historical audit information, as well as to allow auditors to sharpen their initial audit determinations.

A. Lexicon for basic audit determinations

For a given alert, exactly one of the following basic determinations (in bold below) must be selected:

- **Complex**: The alert is too difficult to judge in a reasonable amount of time and effort. Each organization should document what is a reasonable amount of time and effort. Preferably, an approximate time spent auditing would be documented with each Complex alert. See Audit Rule 2 for more information.
- **Dependent**: The alert indicates a code flaw which could only be true if a different coding flaw that is marked True or "Dangerous construct" occurs earlier in the code execution. See Audit Rule 3 for more information. This determination should include a reference to the related alert, via the alert's unique identifier.
- **False**: The code in question does *not* violate the condition indicated by the alert.
- **True**: The code in question violates the condition indicated by the alert.
- **Unknown**: (default value before auditing) None of the above.

B. Lexicon for supplemental audit determinations

For a given alert, any of the following labels (in bold below) may be added, as appropriate. They may be applied to the alert in addition to its basic determination.

- **Dangerous construct**: This label is accompanied with a risk level (High, Medium, or Low). The precise notion of risk is left up to the organization. One example use is that alerts marked "False" may be marked "Dangerous" to indicate the code requires attention despite not violating the indicated condition. See Audit Rule 10 for more information.
- **Dead**: The code in question is unused and is not exportable. See Audit Rule 4 for more information.
- **Ignore**: The code in question does not require mitigation. There are many reasons an organization may want to ignore an alert. For example, an organization may decide to ignore all alerts in a module that they intend to replace.
- **Inapplicable environment**: The alert is not True in the current environment (operating system platform, hardware, CPU), but in a more suitable environment the alert could be True. SA tools occasionally give alerts that are not applicable to a particular platform because some developers want to maximize their code portability. See Audit Rule 7 for more information.

An auditing system should allow an auditor to create a free-form note about each alert which provides information for the developers, future auditors of the same section of code, and developers. Passing information using notes (e.g., details about non-obvious code flaws) saves other workers time when they deal with this section of code.

An organization may choose to add their own supplemental labels (although the basic labels should not be modified). This would be useful for handling alerts that receive treatment that is specific to an organization.

For example, it has been reported that one useful label would be "Misconfigured". This label would apply to an alert that was produced by an analysis tool that was misconfigured or improperly run. An organization might choose to identify such alerts in an initial pass before proceeding with the main audit. Such an organization could add "Misconfigured" as a supplemental label, and they would be responsible for dictating how and when to apply it. Alternatively, the organization could mark such alerts as "Ignore" and add a note explaining why. The note might include the term "Misconfigured", which could be automatically queried to produce all such alerts if desired.

C. Lexicon for additional auditing terms

More terms for the auditing lexicon are listed below, in bold.

- **Alert**: A warning, typically from an SA tool, which indicates a violation of a well-defined *condition*. Many SA tools' alerts directly map the code flaw to a coding

taxonomy. Alerts are assumed to possess a unique identifier, for the purpose of reference. Auditors can create alerts manually when expedient.

- **Message**: Text associated with an alert that describes it. Alerts may have multiple messages.
- **Audit Determination**: Decision made by human reviewer about the validity of an SA tool's alert. This may be made with respect to only a message from the SA tool itself, or with respect to a coding taxonomy that the alert is mapped to. The audit record should maintain information about which coding taxonomy conditions (or direct SA tool alert) that the determination was based on.
- **Exportable Code**: Code that is made available to external applications, e.g., part of a public API, shared library, or public class.
- **Condition**: A constraint or property of validity with which code should comply. SA tools try to detect if code violates conditions.
- **Coding Taxonomy**: A named set of coding rules, weaknesses, standards, or guidelines. See Section III for example taxonomies. Each rule or weakness is considered a single condition. An SA tool's alert may include a mapping to conditions in one or more coding taxonomies (e.g., MITRE's "CWE-Compatible Products and Services" [15] lists many static analysis tools that provide CWE output). Frameworks for handling alerts may separately provide mappings between alerts and coding taxonomies (e.g., SCALe [16], ThreadFix [3], and CodeDX [17]).

D. Basic and supplemental determinations

An alert has exactly one basic determination, and any number of applicable supplemental determinations. We categorized determinations as basic or supplemental by considering all possible pairs. Mutually exclusive determinations were categorized as basic, the remainder as supplemental. Based on these determinations, an organization can make informed allocation of resources (e.g., auditor time and developer time). An auditor might stop auditing an alert after making any determination (basic or supplemental).

The Unknown determination allows organizations to estimate the remaining amount of work for an audit, e.g., by counting the number of Unknown alerts.

True and False help an organization focus on actual code flaws. For example, developer time may only be allocated for True alerts. "Complex" allows identification of expensive (with respect to auditor time) alerts. Analysis of Complex alerts can be put on hold until resources are available. The "Dangerous Construct" severity can be used to prioritize flaws to be repaired first. An organization might decide to mark certain alerts or code constructs as "Dangerous Construct", without investing the effort to make other determinations. For high-severity Dangerous alerts, an organization might choose to do code repairs without finding if a true code flaw exists. "Inapplicable environment" indicates alerts that do not impact the software given its current set of target platforms. Organizations may choose to ignore such alerts, or to address them if targets change. Also, this marking could help an organization to estimate the cost of supporting an additional platform.

An auditor may already know some code is unreachable, and alerts in that code can be marked as "Dead". To save auditor time, some organizations choose not to further analyze alerts after they are marked Dead. Other times, determination that code is dead may require whole-program analysis. Often an auditor can mark code as one of [True, False, or Dependent], before sufficient analysis exists to mark it Dead. An organization may decide not to use further auditor time on the alert to decide if code is Dead.

Code that is Dependent usually does not require further attention, because dependent code is usually fixed as the code it depends on is also fixed. However, an auditor may indicate that Dependent code still requires individual attention by marking it Dangerous. Marking an alert "Ignore" indicates that it mandates no further attention. This is most useful when paired with alerts whose basic determination indicates otherwise (e.g., True or Dependent). An organization might also apply Ignore to False or other alerts, especially if they use an automated tool to mark alerts with other characteristics (e.g., such as being in an obsolete module).

Figure 1 shows transitions between basic determinations that could be made for a single alert as an auditor studies the code and her understanding improves, assuming she makes no auditing errors and the code does not change. From any state, an alert may move to any state to the right (from Unknown to True, for example), but may not move to the left (e.g., from False to Complex). An alert that transitioned from False to Complex would indicate either a code change or that an auditor made an error in marking it False.

III. Auditing Rules

Auditing rules were chosen with the goal of clarifying ambiguous auditing scenarios, to make audit determinations consistent. These rules are intended to apply to auditing by all organizations. For example, setting a zero-new-defect policy would not be affordable for some organizations. As a standard universal auditing rule, that type of auditing rule would not suffice. However, an organization could support such a policy with their auditors and their developers who do code repair, by using our suggested lexicon (and possibly adding a supplemental determination of "New Alert" that an analysis tool could easily automatically mark).

The following auditing rules are designed to be independent of programming language, platform, and coding taxonomy. Examples of taxonomies that can be used with the auditing rules include MITRE's Common Weakness Enumeration [18], the SEI CERT Coding Standards for C [19] or Java [20], and MISRA rules [21]. Alternatively, these auditing rules can be used with no taxonomy at all, in which case each SA tool is expected to endow its alerts with sufficient documentation of the conditions it warns about.

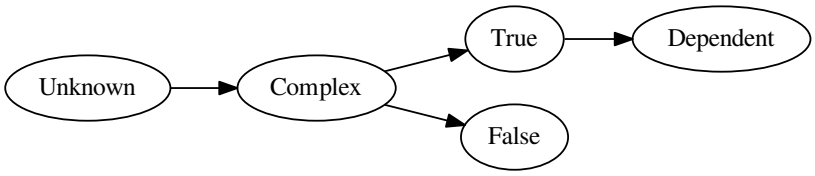

Fig. 1. Basic determination changes are transitive, if correct and no code changes

```
static int jas_icccurv_input(jas_iccattrval_t *
    attrval, jas_stream_t *in, int cnt)
{
    jas_icccurv_t *curv = &attrval->data.curv;
    // ...
    if (JAS_CAST(int, 4 + 2 * curv->numents) != cnt
    )
        goto error;
}
```

Fig. 2. Example for Rule 1 from JasPer (`jas_icc.c`)

```
jas_image_t *jas_image_copy(jas_image_t *image)
{
    jas_image_t *newimage;
    int cmptno;
    newimage = jas_image_create0();
    if (jas_image_growcmpts(newimage, image->
    numcmpts_)) {
        goto error;
    }
    for (cmptno = 0; cmptno < image->numcmpts_; ++
    cmptno) {
        if (!(newimage->cmpts_[cmptno] =
    jas_image_cmpt_copy(image->cmpts_[cmptno]))) {
            goto error;
        }
        ++newimage->numcmpts_;
    }
}
```

Fig. 3. Example for Rule 3 from JasPer (`jas_image.c`)

To illustrate auditing rule use, we provide code examples in the C language. Most of the code examples accompanying these rules come from the open-source JasPer Project [22].

1) *Assume external inputs to the program are malicious.*
In applying this rule, also assume external inputs to arguments of exportable functions are malicious.

Example: The code in Fig. 2 produces the following alert for the highlighted section: Rule INT31-C (Ensure that integer conversions do not result in lost or misinterpreted data). The code reads an unsigned 32-bit integer which comes from the `attrval` function parameter. The goal of this software is to parse an image file for various purposes. The input comes from an external file, which we assume can be designed by a malicious attacker. Therefore we must assume that the integer along with the remainder of the input is untrusted. The function then performs some arithmetic on the integer and casts it to an int. This definitely has the potential for misinterpreted values. This can cause `4 + 2 * curv->numents` to be greater than `INT_MAX`. The casting in the `JAS_CAST()` macro changes signedness. Therefore, this alert should be marked as True.

2) *Some alerts are too difficult to judge; they should be marked Complex.*
Theoretically, an auditor can mark every alert as True or False if they have complete access to the source code and sufficient time to fully understand how the code works. Unfortunately, auditors have limited time and understanding, and are occasionally unable to make a thorough investigation in a reasonable period of time. In such a case, the auditor should mark the alert Complex. The auditor is free to reexamine the alert later and mark it True or False if they can make a sound judgment. They may also refer the alert to an auditing expert, who may be able to render a judgment. An auditor could finish a large audit quickly by marking every alert Complex, but then the audit would be useless. Therefore, auditors should spend a reasonable amount of time analyzing each alert, and only mark it Complex if a determination cannot be made in that time. Each organization is responsible for deciding what constitutes a "reasonable amount of time". A suggested initial value is 5 minutes per alert [23].

3) *If an alert is true only when a previous alert is true, mark it Dependent.*
Causal relationships may exist between alerts. Unless these relationships are explicitly identified by the tool, they may be difficult to determine during the audit process, especially if the alerts are far apart in the source code. In the absence of prior knowledge about alert relationships, an auditor should proceed as if alerts are causally unrelated.

However, if an auditor is able to deduce a causal relationship between alerts during their audit, they should note this relationship. In particular, if an auditor is convinced that an alert (alert A) can only be true if a different true alert (alert B) occurs on all control flow paths leading to alert A, then alert A should be marked as Dependent. See Rule 3 for how to handle the earlier violation (alert B). Ideally, only the root cause of a set of alerts should be marked True, and other alerts in the set should be marked Dependent. This helps to target code repair work more precisely.

Each Dependent marking should include a reference to the related alert, using its unique identifier.

Example: Code in Fig. 3 generates three alerts, one for

each highlighted section; each claims that the code violates Rule EXP34-C (Do not dereference null pointers). We know that the `jas_image_create0()` call before the first alert can return null or a non-null value (a return value from `malloc()`). We also know that the `jas_image_growcmpts()` call in the first alert dereferences `newimage`. It is also clear that a single fix can eliminate all three EXP34-C alerts. Adding a null check for `newimage` after `jas_image_create0()` returns would guarantee that `newimage` is not null for the rest of the function. The first alert is indisputably True, since a null dereference is possible in the callee. It can be argued that the subsequent alerts should be False, because if `newimage` is null, the program crashes due to the initial null dereference. It can also be argued that the subsequent alerts should be True, because they make the same mistake of dereferencing `newimage`. Without formal auditing rules, auditors can make inconsistent judgements for these alerts. Using this rule, auditors should mark these two subsequent alerts as Dependent.

4) *Handle an alert in unreachable code depending on whether it is exportable.*

Unreachable code, also called dead code, is code that is never invoked. In a single program, dead code is a liability, as it might still be invoked by an attacker if they obtain code execution privileges, even in some restricted context, such as in a Java security sandbox. Dead code may also be indicative of a more serious, underlying issue, such as a logic error in the program (e.g., a tautological predicate). Return-oriented programming is one exploitation technique that can take advantage of dead code [24]. Many organizations wish to identify dead code and either remove it or fix the logic errors that result in the code being dead. Therefore this unique audit marking allows one to readily identify dead code. For example, CERT provides recommendation MSC12-C (Detect and remove code that has no effect or is never executed). Such code is considered *non-exportable dead code*. An alert in non-exportable dead code should be marked as Dead.

Many libraries will provide a public API of functions, and programs using these libraries need not invoke every function. Such a function, while it may be considered dead code within a program, may be invoked by a separate program. Such exportable code is to be handled differently than non-exportable dead code. An alert in exportable dead code should be audited by assuming untrusted inputs to the function, as discussed in Auditing Rule 1.

Example: In Fig. 4, the variable `always_zero` is initialized to the value of 0, and it is not set to any other value in this function. Therefore, the comparison of this variable to 1 is always False, so the highlighted line has an alert in unreachable non-exportable code. Thus, this alert should be marked as Dead.

5) *An alert might indicate a true violation of the condition it is mapped to, even if the alert's message is useless or incorrect.* While an alert message can be informative, it should not solely determine whether the alert is true or false. The message may be correct, yet the alert may be False. Conversely, the message may be incorrect, yet the alert may be True. The alert message should be taken as a hint, but the auditor may ignore the message when considering an alert. This is especially useful if the auditor can rely on a coding taxonomy.

```
int func()
{
  int sometimes_zero = 1;
  int always_zero = 0;
  int returnValue = 55;

  sometimes_zero = returnAnInt();
  if (always_zero == 1) {
    returnValue = performIfOne();
  } else {
    returnValue = performIfNotOne();
  }
  return(returnValue);
}
```

Fig. 4. Example for Rule 4

6) *Mark an alert True even if code maintainers will protest.* There can be tension between code maintainers who wish to avoid superfluous changes and code auditors who desire code security. Often, maintainers will understand the code more thoroughly than auditors, and they may have the final say on whether code is modified due to an auditor's recommendations. Auditors should mark true alerts as True, even if maintainers may protest the determination.

For example, suppose that C code accesses memory 0 (e.g., dereferences null) because it runs only on a platform that allows this. The auditor should still report a potential null pointer dereference, to make the organization aware of the issue (e.g., the organization might choose to resolve the problem by documenting platform requirements).

This rule permits auditors to ignore how the organization handles an alert's determinations and to focus solely on the correctness of the code.

7) *Unless instructed otherwise, assume code must be portable.*

Sometimes an auditor must audit a code base without knowledge of the target architecture. When auditing alerts from this code base, it is reasonable to err on the side of portability. If a diagnosed segment of code malfunctions on certain platforms, and in doing so violates a condition, this is suitable justification for marking the alert True.

For alerts for any codebase, the auditor might answer one of two questions: 1. Is this code secure for one particular platform (e.g., 32-bit x86)?; or 2. Is this code secure for all platforms (e.g., strictly conformant to the ISO C standard)?

Which question to answer often depends on the codebase, but it applies to all alerts within that codebase. If an auditor is not given this information, the auditor should assume that the codebase is intended to be portable; that is, it should run on any platform that supports the programming language.

8) *When auditing an alert, if a second true violation is discovered, its alert should be marked True.*

Auditors often discover defects in the code that are distinct from a particular alert they are examining. For example, the line of code in question may violate a condition not indicated by the alert. Or the defect might involve a different line of code entirely.

Some auditors may restrict the set of conditions they are auditing against. This is useful for auditors in training, or in scanning code they have not audited before. If the condition violated by any defect is not one of the conditions the auditor has been instructed to focus on, the auditor is instructed to ignore the defect. Otherwise, the auditor should, in addition to the current alert, produce an alert associated with the defect and mark it True.

The auditor may search to see if any tool already reported an alert associated with the defect in question, and if a suitable alert is found, it should be marked True. If no tool reports a suitable alert, the auditor should manually create an alert describing the defect.

Discovery of a distinct defect while auditing an alert does not warrant automatically marking the original alert as True, even if the distinct defect involves the same line of code. Each distinct defect merits an independent alert. If alerts are mapped to a coding taxonomy, the new alerts should likewise also be mapped to the same taxonomy.

9) *Auditors must understand the language and the current alert's condition.*

A poor understanding of the programming language will render many edge cases inscrutable, and many alert audits depend on these edge cases. To correctly evaluate an alert, an auditor must be familiar with its associated condition.

If no taxonomy is being used, the condition is implied by the tool's alert itself. The auditor must understand their analysis tool, and correct configuration and runtime options of the static analysis tool can greatly affect results. If the condition is in a taxonomy, a deep understanding of the alert and SA tool can be replaced by an understanding of the taxonomy condition.

We recommend that new auditors focus on one condition (e.g., null pointer dereferences) and audit many alerts just for that one condition, before attempting a second condition. They should then do the same for subsequent conditions they need to audit, to best learn each condition. Access to the authoritative standard for the programming language also helps.

10) *Do not arbitrarily extend the scope of a condition.*

When auditing an alert, determine only if the precise condition is violated. If code is judged insecure but not in violation of the condition in question, it may violate a different condition being audited for. Even if it does not violate any conditions in the coding taxonomy, the code may still warrant attention. Rather than marking such an alert True, an auditor could mark it Dangerous (see "Dangerous construct" in Section III lexicon), along with a note explaining the issue.

Example: The code in Fig. 5 produces the following alert for the highlighted section: Rule MEM35-C (Allocate sufficient memory for an object). This code will behave unexpectedly, because it performs a multiplication in a `sizeof` expression, which will discard its product. This violates CERT rule ARR38-C (Guarantee that library functions do not form invalid pointers); see Rule 8 for how to handle the ARR38-C issue. However, this is not a violation of MEM35-C, which deals strictly with memory allocation, and no memory is allocated in this code. Therefore, the auditor should mark this alert as False, despite the code's other problems.

```
static jpc_enc_tcmpt_t *tcmpt_create(
    jpc_enc_tcmpt_t *tcmpt, jpc_enc_cp_t *cp,
    jas_image_t *image, jpc_enc_tile_t *tile)
{
    // ...
    memset(tcmpt->stepsizes, 0, sizeof(tcmpt->
    numstepsizes * sizeof(uint_fast16_t)));
    // ...
}
```

Fig. 5. Example for Rule 10 from JasPer (`jpc_enc.c`)

```
int jas_tvparser_next(jas_tvparser_t *tvp)
{
  char *p;
  char *tag;
  char *val;

  /* Skip any leading whitespace. */
  p = tvp->pos;

    /* Is a value field not present? */
    if (*p != '=') {
      if (*p != '\0' && !isspace(*p)) {
        return -1;
      }
```

Fig. 6. Example for Rule 11 from JasPer (`jas_tvp.c`)

11) *Code that behaves as expected might still violate a condition.*

Do not assume code is secure simply because it seems to work. Vulnerabilities get discovered in production code.

Example: The code in Fig. 6 produces the following alert: Line 165, Rule STR34-C (Cast characters to unsigned char before converting to larger integer sizes). This rule dictates that `plain` character pointers should be cast to unsigned char before being passed to `isspace()`. This code may work properly on an x86 platform, but would have unexpected behavior on a platform where plain chars are implemented as `unsigned ints` (they are `signed ints` on x86). This example also supports Rule 7, which dictates that the auditor must assume code is portable, unless instructed otherwise. If the auditor has been instructed to assume this code only runs on x86, then they would mark this False. Otherwise, they should mark this alert True.

12) *Multiple messages help in understanding an alert.*

Many SA tools produce multiple messages for a single overarching alert. The tool might provide a trace through a function that illustrates why some code may be in error. It is worthwhile to trace through multiple messages to understand the tool's rationale. Similarly, multiple SA tools may report violation of the same condition on the same line of the same file, as may be seen in multi-tool frameworks (SCALe [16], CodeDX [17], Threadfix [3]). Messages can vary between tools, and viewing all the messages may be helpful in auditing these alerts.

IV. Small Test Case

Two of the three collaborating organizations on this project used the auditing rules for auditing alerts from their own codebases. Using the auditing rules, one of the collaborators audited 93 alerts for 8 CERT C-language coding rules, and the other collaborator audited 195 alerts for 15 CERT Java-language coding rules. Rule 8 was developed to clarify how an alert determination should be handled, due to an issue discovered during an audit by a collaborator. The full auditing lexicon was not used, but it was developed partially as a result of discussions with the collaborators.

V. Conclusions & Future Work

The software engineering community needs a widely-accepted standard lexicon and set of rules for auditing static analysis alerts. This would guide different auditors to make the same determination for an alert. This paper provides a suggested lexicon and auditing rules, detailing rationales based on modern software engineering practices for each term in the lexicon and for each rule. In future work, we plan to gather more extensive feedback on the lexicon and rules via surveys, focus groups, and communication with subject matter experts, to improve the lexicon and rules. We will also implement an auditing system with all the audit determinations described in the lexicon, and work with collaborators to extensively test utility of the rules and lexicon on real-world code.

Acknowledgment

We thank the anonymous reviewers for their comments. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. Carnegie Mellon® and CERT® are registered marks of Carnegie Mellon University. DM-0003958

References

[1] A. Shunn, C. Woody, R. Seacord, and A. Householder, "Strengths in Security Solutions," 2013.

[2] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security & Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[3] Threadfix, "ThreadFix," http://www.threadfix.it/, accessed June 23, 2016.

[4] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: an experimental approach," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 341–350.

[5] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis." in *Usenix Security*, vol. 2013, 2005.

[6] A. Delaitre, V. Okun, and E. Fong, "Of massive static analysis data," in *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*. IEEE, 2013, pp. 163–167.

[7] V. Ciriello, G. Carrozza, and S. Rosati, "Practical experience and evaluation of continuous code static analysis with C++ test," in *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation*. ACM, 2013, pp. 19–22.

[8] B. Carlsson and D. Baca, "Software security analysis-execution phase audit," in *31st EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2005, pp. 240–247.

[9] C. Cifuentes and N. Keynes, "Internal Deployment of the Parfait Static Code Analysis Tool at Oracle," in *Asian Symposium on Programming Languages and Systems*. Springer, 2013, pp. 172–175.

[10] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[11] B. Chess and J. West, *Secure programming with static analysis*. Pearson Education, 2007.

[12] "Prioritizing Alerts from Static Analysis to Find and Fix Code Flaws," https://insights.sei.cmu.edu/sei_blog/2016/06/prioritizing-alerts-from-static-analysis-to-find-and-fix-code-flaws.html, 2016, accessed: 2016-06-27.

[13] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, 2011.

[14] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6. ACM, 2004, pp. 83–93.

[15] MITRE, "CWE-Compatible Products and Services," https://cwe.mitre.org/compatible/compatible.html, accessed June 23, 2016.

[16] D. Plakosh, R. Seacord, R. W. Stoddard, D. Svoboda, and D. Zubrow, "Improving the Automated Detection and Analysis of Secure Coding Violations," 2014.

[17] *CodeDx*, http://codedx.com/, accessed June 23, 2016.

[18] MITRE, "Common Weakness Enumeration: A Community-Developed Dictionary of Software Weakness Types," https://cwe.mitre.org, accessed June 22, 2016.

[19] R. C. Seacord, *The CERT C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems*. Pearson Education, 2014.

[20] F. Long, D. Mohindra, R. C. Seacord, D. Sutherland, and D. Svoboda, *The CERT Oracle Secure Coding Standard for Java*. Pearson Education, 2012.

[21] MISRA, *Guidelines for the Use of the C Language in Critical Systems*. MISRA, March 2013, iSBN 978-1-906400-10-1.

[22] "JasPer," https://www.ece.uvic.ca/~frodo/jasper, accessed June 23, 2016.

[23] M. Hayward, "Static analysis vulnerabilities and defects: Best practices for both agile and waterfall development environments," Dec 2008, White Paper.

[24] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.