Predicting Vulnerable Software Components

Stephan Neuhaus* Thomas Zimmermann+ Christian Holler* Andreas Zeller*

* Department of Computer Science
Saarland University, Saarbrücken, Germany
{neuhaus,holler,zeller}@st.cs.uni-sb.de

+ Department of Computer Science
University of Calgary, Calgary, Alberta, Canada
tz@acm.org

ABSTRACT

Where do most vulnerabilities occur in software? Our Vulture tool automatically mines existing vulnerability databases and version archives to map past vulnerabilities to components. The resulting ranking of the most vulnerable components is a perfect base for further investigations on what makes components vulnerable.

In an investigation of the Mozilla vulnerability history, we surprisingly found that components that had a single vulnerability in the past were generally not likely to have further vulnerabilities. However, components that had similar imports or function calls were likely to be vulnerable.

Based on this observation, we were able to extend Vulture by a simple predictor that correctly predicts about half of all vulnerable components, and about two thirds of all predictions are correct. This allows developers and project managers to focus their their efforts where it is needed most: "We should look at `nsXPInstallManager` because it is likely to contain yet unknown vulnerabilities."

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Statistical methods*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software.*

General Terms: Security, Experimentation, Measurement

Keywords: Software Security, Prediction

1. INTRODUCTION

Many software security problems are instances of general patterns, such as buffer overflow or format string vulnerabilities. Some problems, though, are specific to a single project or problem domain: JavaScript programs escaping their jails are a problem only in web browsers. To improve the security of software, we must therefore not only look for general problem patterns, but also learn *specific* patterns that apply only to the software at hand.

Modern software development usually does a good job in tracking past vulnerabilities. The Mozilla project, for instance, maintains a vulnerability database which records all incidents. However, these databases do not tell how these vulnerabilities are distributed across the Mozilla codebase. Our *Vulture* tool automatically mines a vulnerability database and associates the reports with the *change history* to map vulnerabilities to individual components (Figure 1).

Vulture's result is a *distribution of vulnerabilities* across the entire codebase. Figure 2 shows this distribution for Mozilla: the darker a component, the more vulnerabilities were fixed in the past.The distribution is very uneven: Only 4% of the 10,452 components were involved in security fixes. This raises the question: *Are there specific code patterns that occur only in vulnerable components?*

In our investigation, we were not able to determine code features such as, code complexity or buffer usage that would correlate with the number of vulnerabilities. What we found, though, was that vulnerable components shared similar sets of *imports* and *function calls.* In the case of Mozilla, for instance, we found that of the 14 components importing `nsNodeUtils.h`, 13 components (93%) had to be patched because of security leaks. The situation is even worse for those 15 components that import `nsIContent.h`, `nsIInterfaceRequestorUtils.h` and `nsContentUtils.h` together—they *all* had vulnerabilities. This observation can be used for automatically *predicting* whether a new component will be

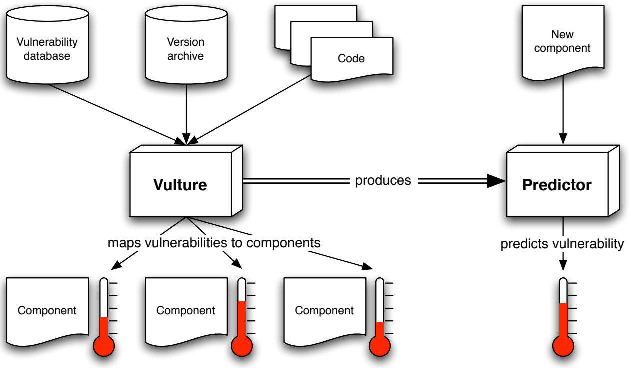


Figure 1: How Vulture works. Vulture mines a *vulnerability database* (e.g. a Bugzilla subset), a *version archive* (e.g. CVS), and a *code base*, and maps past vulnerabilities to components. The resulting *predictor* predicts the *future vulnerabilities of new components,* based on their imports or function calls.

CCS'07, October 29–November 2, 2007, Alexandria, Virginia, USA.

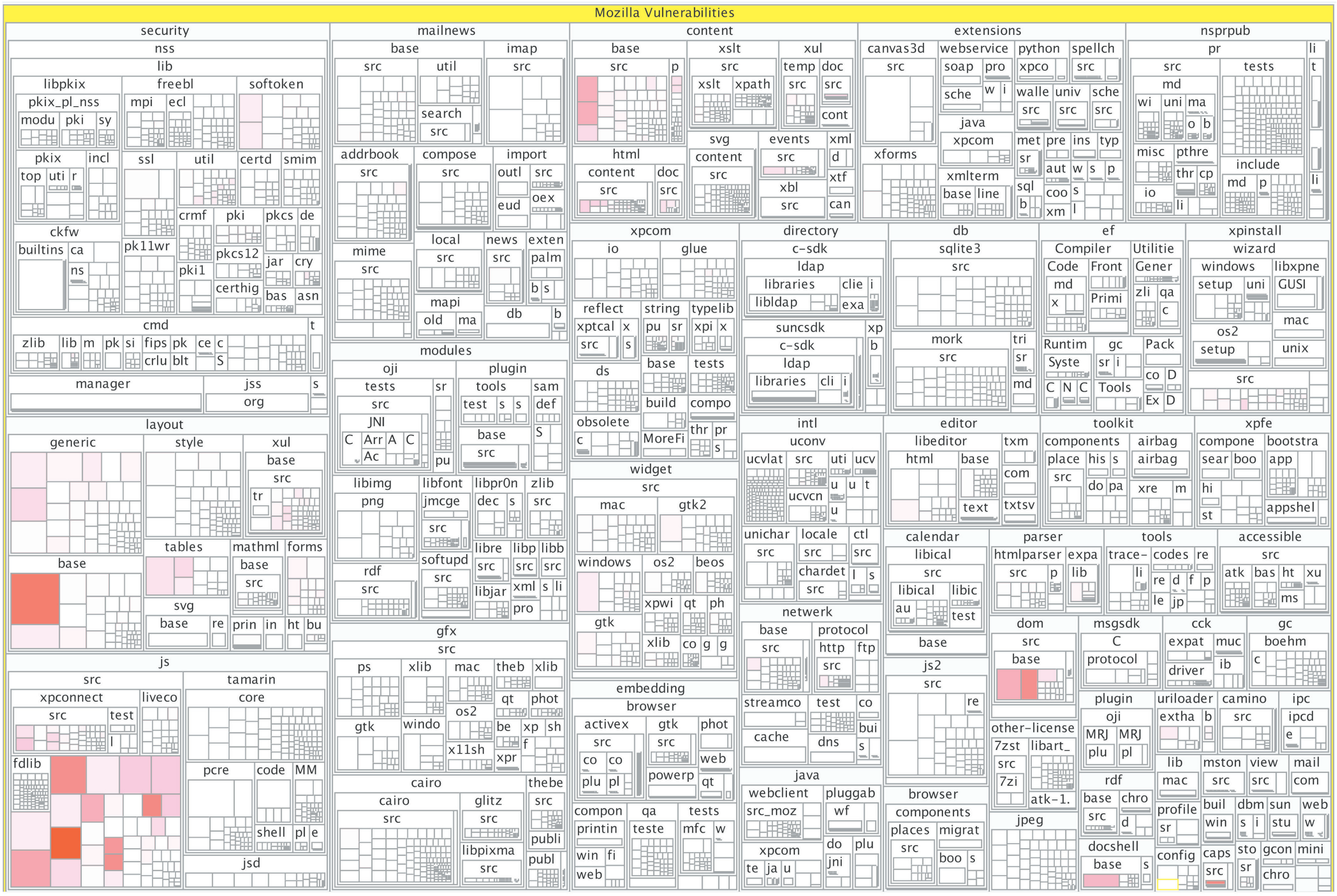


Figure 2: Distribution of vulnerabilities within Mozilla's codebase. A component's area is proportional to its size; its shade of gray is proportional to its number of vulnerabilities. A white box means no vulnerabilities, as is the case for 96% of the components.

vulnerable or not: "Tell me what you import or what you call, and I'll tell you how vulnerable you are."

After discussing the scope of this work (Section 2), the remainder of this paper details our original contributions, which can be summarized as follows.

- We present a fully automatic way of mapping vulnerabilities to components (Section 3).
- We provide empirical evidence that vulnerabilities correlate with component imports (Section 4).
- We show how to build fully automatic predictors that predict vulnerabilities of new components based on their imports and function calls (Section 5).
- Our evaluation on the Mozilla project shows that these predictors are accurate (Section 6).

After discussing related work (Section 7), we close with conclusions and future work (Section 8).

2. SCOPE OF THIS WORK

Our work is empirical and statistical: we look at correlations between two phenomena—vulnerabilities on one hand and imports or function calls on the other—, but we do not claim that these are cause-effect relationships. It is clearly not the case that importing some import or calling some function *causes* a vulnerability. Programmers writing that import statement or function call generally have no choice in the matter: they need the service provided by some import or function and therefore have to import or call it, whether they want to or not.

Our hypothesis is that the cause of the vulnerability is the import's or function's *domain*, that is, the range of services that it uses or implements. It appears that some domains are more risky than others, and being associated with a particular domain increases the risk of having a vulnerability. Different projects might have different risky domains, which would lead Vulture to mine project-specific vulnerability patterns.

We have also identified the following circumstances that could affect the validity of our study:

Study size. The correlations we are seeing with Mozilla could be artifacts that are specific to Mozilla. They might not be as strong in other projects, or the correlations might disappear altogether. From our own work analyzing Java projects, we think this is highly unlikely [29]; see also Section 7 on related work.

Bugs in the database or the code. The code to analyze the CVS or import the Security Advisories into the database could be buggy; the inputs to the machine-learning methods or the code that assesses the effectiveness of these methods could be wrong. All these risks were mitigated either by sampling small subsets and checking them manually for correctness, or by implementing the functionality a second time starting from scratch and comparing the results. For example, some machine-learning inputs were manually checked, and the assessment code was rewritten from scratch.

Bugs in the R library. We rely on a third-party R library for the actual computation of the SVM and the predictions [9], but this library was written by experts in the field and has undergone cross-validation, also in work done in our group [29].

Wrong or noisy input data. It is possible that the Mozilla source files contain many "noisy" import relations in the sense that some files are imported but actually never used; or the Security Advisories that we use to map vulnerabilities to components could accidentally or deliberately contain wrong information. Our models do not incorporate noise. From manually checking some of the data, we believe the influence of noise to be negligible, especially since results recur with great consistency, but it remains a (remote) possibility.

Yet unknown vulnerabilities. Right now, our predictions are evaluated against known vulnerabilities in the past. Finding future vulnerabilities in flagged components would improve precision and recall; finding them in unflagged components would decrease recall.

3. COMPONENTS AND VULNERABILITIES

3.1 Components

For our purposes, a *component* is an entity in a software project that can have vulnerabilities. For Java, components would be `.java` files because they contain both the definition and the implementation of classes. In C++, and to a lesser extent in C, however, the implementation of a component is usually separated from its interface: a class is declared in a header file, and its implementation is contained in a source file. A vulnerability that is reported only for one file of a two-file component is nevertheless a vulnerability of the entire component. For this reason, we will combine equally-named pairs of header and source files into one component.

In C, it is often the case that libraries are built around abstractions that are different from classes. The usual case is that there is one header file that declares a number of structures and functions that operate on them, and several files that contain those functions' implementations. Without a working build environment, it is impossible to tell which source files implement the concepts of which header file. Since we want to apply Vulture to projects where we do not have a working build environment—for example because we want to analyze old versions that we cannot build anymore due to missing third-party software—, we simply treat files which have no equally-named counterpart as components containing just that file. We will subsequently refer to components without any filename extensions.

Of course, some components may naturally be self-contained. For example, a component may consist only of a header file that includes all the necessary implementation as inline functions there. Templates must be defined in header files. A component may also not have a header file. For example, the file containing a program's *main* function will usually not have an associated header file. These components then consist of only one file.

3.2 Mapping Vulnerabilities to Components

A *vulnerability* is a defect in one or more components that manifests itself as some violation of a security policy. Vulnerabilities are announced in security advisories that provide users workarounds or pointers to fixed versions and help them avoid security problems. In the case of Mozilla, advisories also refer to a bug report in the Bugzilla database. We use this information, to map vulnerabilities to components through the fixes that remove the defect.

First we retrieve all advisories from the Web to collect the defects, in case of Mozilla from the "Known Vulnerabilities in Mozilla Products" page.[1] We then search for references to the Bugzilla database that typically take the form of links to its web interface:

```
https://bugzilla.mozilla.org/show_bug.cgi?id=362213
```

The number at the end of this URL is the *bug identifier* of the defect that caused the vulnerability. We collect all bug identifiers and use them to identify the corresponding fixes in the version archive. In version archives every change is annotated with a message that describes the reason for that change. In order to identify the fixes for a particular defect, say 362213, we search these messages for bug identifiers such as "362213", "Bug #362213", and "fix 362213" (see also Figure 3). This approach is described in detail by Śliwerski et al. [30] and extends the approaches introduced by Fischer et al. [10] and by Čubranić et al. [7].

Once we have identified the fixes of vulnerabilities, we can easily map the names of the corrected files to components. Note that a security advisory can contain several references to defects, and a defect can be fixed in several files.

It is important to note that we do not analyze binary patches to programs, but source code repository commits. Binary patches usually address a number of bugs at once, which are not necessarily vulnerabilities, or contain functionality enhancements. In contrast, commits are very specific, fixing only one vulnerability at a time. This is why we can determine the affected components with confidence.

3.3 Vulnerable Components in Mozilla

Mozilla as of 4 January 2007 contains 1,799 directories and 13,111 C/C++ files which are combined into 10,452 components. There were 134 vulnerability advisories, pointing to 302 bug reports. Of all 10,452 components, only 424 or 4.05% were vulnerable.

Security vulnerabilities in Mozilla are announced through Mozilla Foundation Security Advisories (MFSAs) since January 2005 and are available through the Mozilla Foundation's web site [33]. These advisories describe the vulnerability and give assorted information, such as Bugzilla bug identification numbers. Of all 302 vulnerability-related bug reports, 280 or 92.7% could be assigned to components using the techniques described above.[2]

[1] http://www.mozilla.org/projects/security/known-vulnerabilities.html

[2] Some bug reports in Bugzilla [32] are not accessible without

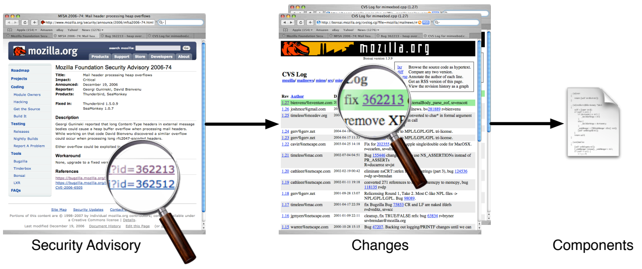


Figure 3: Mapping Mozilla vulnerabilities to changes. We extract bug identifiers from security advisories, search for the fix in the version archive, and from the corrected files, we infer the component(s) affected by the vulnerability.

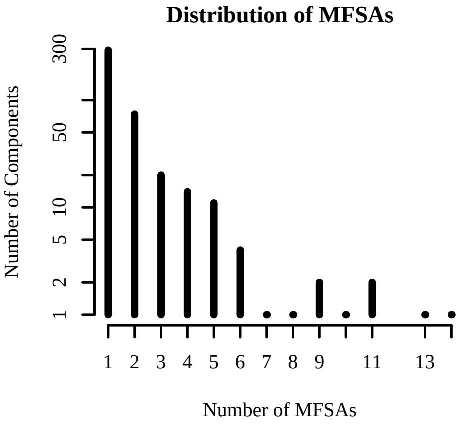


Figure 4: Distribution of Mozilla Foundation Security Advisories (MFSAs). The y axis is logarithmic.

If a component has a vulnerability-related bug report associated with it, we call it *vulnerable*. In contrast to a vulnerable component, a *neutral* component has had no vulnerability-related bug reports associated with it so far.

The distribution of the number of MFSAs can be seen in Figure 4. The most important result from this histogram is that it directly contradicts an item of security folklore that says that components that had vulnerabilities in the past will likely have vulnerabilities in the future. If that were truly the case, the histogram should show ascending numbers of components with ascending numbers of reports. In fact, however, the opposite is true: there were twice as many components with one MFSA (292) than all components with two or more MFSAs *combined* (132).

One consequence of this empirical observation is that the number of past vulnerability reports is not a good predictor for future reports, because it would miss all the components that have only one report. Indeed, when we take the CVS from July 24, 2007—encompassing changes due to MFSAs 2007-01 through 2007-25—we find that 149 components were changed in response to MFSAs. Of these newly fixed components, 81 were repeat offenders, having at least one vulnerability-related fix before January 4. The remaining 68 components had never had a security-related fix.

Rank	Component	SAs	BRs
# 1	`dom/src/base/nsGlobalWindow`	14	14
# 2	`js/src/jsobj`	13	24
# 3.5	`js/src/jsfun`	11	15
# 3.5	`caps/src/nsScriptSecurityManager`	11	15
# 5	`js/src/jsscript`	10	14
# 6	`dom/src/base/nsDOMClassInfo`	9	10
# 7	`docshell/base/nsDocShell`	9	9
# 8	`js/src/jsinterp`	8	14
# 9	`content/base/src/nsGenericElement`	7	10
# 10	`layout/base/nsCSSFrameConstructor`	6	17

Table 1: The top ten most vulnerable components in Mozilla, sorted by associated Mozilla Foundation Security Advisories (SAs) and bug reports (BRs). Components with equal numbers of SAs get an averaged rank.

As for using other metrics such as lines of code and so on to predict vulnerabilities, studies by Nagappan et al. have shown that there is no single metric that correlates with failures across all considered projects [21].

The top ten most vulnerable components in Mozilla are listed in Table 1. The four most vulnerable components all deal with *scripting* in its various forms:

1. `nsGlobalWindow`, with fixes for 14 MFSAs and 14 bug reports, has, among others, a method to set the status bar, which can be called from JavaScript and which will forward the call to the browser chrome.

2. `jsobj` (13 MFSAs; 24 bug reports) contains support for JavaScript objects.

3. `jsfun` (11 MFSAs; 15 bug reports) implements support for JavaScript functions.

4. `nsScriptSecurityManager` (11 MFSAs; 15 bug reports) implements access controls for JavaScript programs.

In the past, JavaScript programs have shown an uncanny ability to break out of their jails, which manifests as a high number of security-related changes to these components.

4. IMPORTS AND FUNCTIONS MATTER

As discussed in Section 3.3, we found that several components related to *scripting* rank among the most vulnerable components. How does a concept like scripting manifest itself in the components' code?

Our central assumption in this work is that what a component does is characterized by its *imports* and *function calls.* A class that implements some form of content—anything that can be in a document's content model—will use functions declared in `nsIContent.h` and will therefore need to import it; a class that implements some part of the Document Object Model (DOM) will likely use functions from—and hence import—`nsDOMError.h`. And components associated with scripting are characterized by the functions from and the import of `nsIScriptGlobalObject.h`.

an authenticated account. We suppose that these reports concern vulnerabilities that have high impact but that are not yet fixed, either in Mozilla itself or in other software that uses the Mozilla codebase. In many cases, we were still able to assign bug reports to files automatically because the CVS log message contained the bug report number. By looking at the diffs, it would therefore have been possible to derive what the vulnerability was. Denying access to these bug reports is thus largely ineffectual and might even serve to alert blackhats to potential high-value targets.

In a strictly layered software system, a component that is located at layer k would import only from components at layer $k+1$; its imports would pinpoint the layer at which the component resides. In more typical object-oriented systems, components will not be organized in layers; still, its imports will include those components whose services it uses and those interfaces that it implements.

If an interface or component is specified in an insecure way, or specified in a manner that is difficult to use securely, then we would expect many components that use or implement that interface or component to be vulnerable. In other words, we assume that it is a component's *domain*, as given by the services it uses and implements, that determine whether a component is likely to be vulnerable or not.

How do imports and function calls correlate with vulnerabilities? For this, we first need a clear understanding of what constitutes an import or a function call and what it means for a set of imports or function calls to be correlated with vulnerability.

In the following discussion, we use the term "feature" to refer to both imports and function calls.

4.1 Imports

In C and C++, a component's *imports* are those files that it references through `#include` preprocessor directives. These directives are handled by the preprocessor and come in three flavors:

`#include <name>` This variant is used to import standard system headers.

`#include "name"` This variant is used to import header files within the current project.

`#include NAME` In this variant, `NAME` is treated as a preprocessor symbol. When it is finally expanded, it must resolve to one of the two forms mentioned above.

The exact computation of imports for C and C++ is difficult because the semantics of the first two variants are implementation-dependent, usually influenced by compile-time switches and macro values. That means that it is not possible to determine exactly what is imported without a working build environment. We therefore adopted the following heuristics:

- We treat every occurrence of `#include` as an import, even though it may not be encountered in specific compile-time configurations—for example because of conditional compilation. The reason is that we want to obtain all possible import relations, not just the ones that are specific to a particular platform.

- We assume that identically-named includes refer to the same file, even though preprocessor directives may cause them to refer to different files. It turns out that this does not happen in Mozilla.

- Implementing the computed include would require a full preprocessor pass over the source file. This in turn would require us to have a fully compilable (or at least preprocessable) version of the project. Fortunately, this use of the include directive is very rare (Mozilla does not use it even once), so we chose to ignore it.

```
#ifdef XP_OS2
  if (DosCreatePipe(&pipefd[0], &pipefd[1], 4096) != 0) {
#else
  if (pipe(pipefd) == -1) {
#endif
    fprintf(stderr, "cannot create pipe: %d\n", errno);
    exit(1);
  }
```

Figure 5: Extract from `nsprpub/pr/tests/sigpipe.c`, lines 85ff. Parsing C and C++ is generally only possible after preprocessing: attempting to parse these lines without preprocessing results in a syntax error.

4.2 Function Calls

In C and C++, a *function call* is an expression that could cause the control flow to be transferred to a function when it is executed.[3] A function call is characterized by the name of the function and a parenthesized list of arguments.

Statically extracting function calls from unpreprocessed C or C++ source code is difficult. Dynamic parsing with type information would require compilable source code and even a full static parsing is blighted by syntax errors caused by some preprocessor statements; see Figure 5. As a consequence, we simply treat all occurrences of *identifier*`(...)` and *identifier*`<...>(...)` as function calls.

Keywords are excluded so that `if` or `while` statements are not erroneously classified as function calls. Also, to match only function calls and not function definitions, these patterns must not be followed by an opening curly bracket. But even with these restrictions, there are many other constructs which match these patterns, such as constructors, macros, forward declarations, member function declarations, initialization lists, and C++ functional-style type casts.

Some of these, like constructors and macros, are very similar to function calls and hence are actually desired. The false classifications of forward declarations, member function declarations, initialization lists, and type casts do not seem to affect our results.

In contrast to these undesirable positive classifications, there are also function calls that are not caught by our heuristic, such as function calls using function pointers or overloaded operators. A simple parser without preprocessing will generally not be able to do type checking, and will therefore not be able to correctly classify such calls. However, we believe that this is a rather uncommon practice in C++, especially in bigger projects such as Mozilla because such dynamic calls are more effectively employed through virtual functions. Hence, we ignore this category of call.

4.3 Mapping Vulnerabilities to Features

In order to find out which feature combinations are most correlated with vulnerabilities, we use frequent pattern mining [1, 18]. The result of frequent pattern mining is a list of feature sequences that frequently occur in vulnerable components. To judge whether these features are significant, we apply the following criteria:

Minimum Support. For imports, the pattern must appear in at least 3% of all vulnerable components. (In other words, it needs a minimum support count of 3%

[3] This cautious phrasing is necessary because of the possibility of inlining.

$P(V\|I)$	$V \wedge I$	$!V \wedge I$	Includes
1.00	13	0	nsIContent.h · nsIInterfaceRequestorUtils · nsContentUtils.h
1.00	14	0	nsIScriptGlobalObject.h · nsDOMCID.h
1.00	19	0	nsIEventListenerManager.h · nsIPresShell.h
1.00	13	0	nsISupportsPrimitives.h · nsContentUtils.h
1.00	19	0	nsReadableUtils.h · nsIPrivateDOMEvent.h
1.00	15	0	nsIScriptGlobalObject.h · nsDOMError.h
0.97	34	1	nsCOMPtr · nsEventDispatcher.h
0.97	29	1	nsReadableUtils.h · nsGUIEvent.h
0.96	22	1	nsIScriptSecurityManager.h · nsIContent.h · nsContentUtils.h
0.95	18	1	nsWidgetsCID.h · nsContentUtils.h

Table 2: Include patterns most associated with vulnerability. The column labeled "Includes" contains the include pattern; the column labeled $P(V|I)$ contains the conditional probability that a component is vulnerable (V) if it includes the pattern (I). The columns labeled $V \wedge I$ and $!V \wedge I$ give the absolute numbers of components that are vulnerable and include the set, and of components that are not vulnerable, but still include the set.

of 424, or 13). For function calls, this threshold is raised to 10%, or 42.

Significance. We only want to include patterns that are more meaningful than their sub-patterns. For this, we test whether the entire pattern is more specific for vulnerabilities than its sub-patterns. Let I be a set of features that has passed the minimum-support test. Then for each proper subset $J \subset I$, we look at all files that feature I and at all files that feature $I - J$. We then classify those files into vulnerable and neutral files and then use the resulting contingency table to compute whether additionally featuring J significantly increases the chance of vulnerability. We reject all patterns where we cannot reject the corresponding hypothesis at the 1% level. (In other words, it must be highly unlikely that featuring J in addition to $I - J$ is independent from vulnerability.)[4]

For patterns that survive these tests, the probability of it occurring in a vulnerable component is much higher than for its subsets. This is the case even though the conditional probability of having a vulnerability when including these particular includes may be small.

4.4 Features in Mozilla

Again, we applied the above techniques to the Mozilla base. In Mozilla, Vulture found 79,494 import relations of the form "component x imports import y", and 9,481 distinct imports. Finding imports is very fast: a simple Perl script goes through the 13,111 C/C++ files in about thirty seconds. We also found 324,822 function call relations of the form "component x calls function y", and 93,265 distinct function names. Finding function calls is not as fast as finding imports: the script needs about 8 minutes to go through the entire Mozilla codebase.

Frequent pattern mining, followed by weeding out insignificant patterns yields 576 include patterns and 2,470 function call patterns. The top ten include patterns are shown in Table 2. Going through all 576 include patterns additionally reveals that some includes occur often in patterns, but not alone. For example, `nsIDocument.h` appears in 45 patterns, but never appears alone. Components that often appear together with `nsIDocument.h` come from directories `layout/base` or `content/base/public`, just like `nsIDocument` itself. Similar observations hold for function call patterns.

Table 2 reveals that implementing or using `nsIContent.h` together with `nsIInterfaceRequestorUtils` and `nsContentUtils.h` correlated with vulnerability in the past. Typical components that imports these are `nsJSEnvironment` or `nsHTMLContentSink`. The first is again concerned with JavaScript, which we already know to be risky. The second has had a problems with a crash involving DHTML that apparently caused memory corruption that could have led to arbitrary code execution (MFSA 2006-64).

Looking at Table 2, we see that of the 35 components importing `nsIScriptSecurityManager.h`, `nsIContent.h`, and `nsContentUtils.h`, 34 are vulnerable, while only one is not. This may mean one of two things: either the component is invulnerable or the vulnerability just has not been found yet. At the present time, we are unable to tell which is true. However, the component in question is `nsObjectLoadingContent`. It is a base class that implements a content loading interface and that can be used by content nodes that provide functionality for loading content such as images or applets. It certainly cannot be ruled out that the component has an unknown vulnerability.

5. PREDICTING VULNERABILITIES FROM FEATURES

In order to predict vulnerabilities from features, we need a data structure that captures all of the important information about components and features (such as which component has which features) and vulnerabilities (such as which component has how many vulnerabilities), but abstracts away information that we consider unimportant (such as the component's name). In Figure 6, we describe our choice: if there are m components and n features, we write each component as a n-vector of features: $\mathbf{x}_k = (x_{k1}, \ldots, x_{kn})$, where for $1 \leq k \leq m$ and $1 \leq j \leq n$,

$$x_{kj} = \begin{cases} 1 & \text{if component } i \text{ features feature } j, \\ 0 & \text{otherwise.} \end{cases}$$

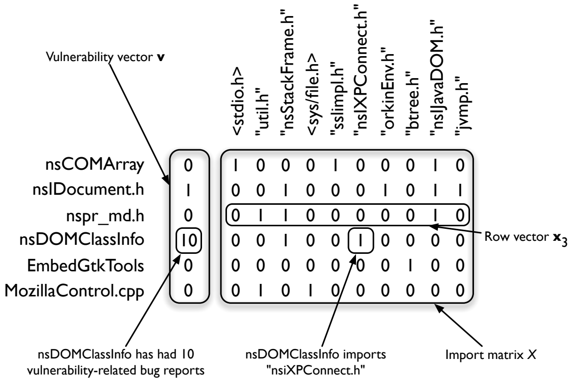
We combine all components into $X = (\mathbf{x}_1, \ldots, \mathbf{x}_m)^t$, the project's *feature matrix.* Entities that cannot contain includes or function calls, such as makefiles, are ignored.

In addition to the feature matrix, we also have the *vulnerability vector* $\mathbf{v} = (v_1, \ldots, v_m)$, where v_j is the number of vulnerability reports associated with component j.

Now assume that we get a new component, $\mathbf{x}_{m+1}$. Our question, "How vulnerable is component $m + 1$?" is now equivalent to asking for the rank of v_{m+1} among the values of $\mathbf{v}$, given $\mathbf{x}_{m+1}$; and "Is component $m + 1$ vulnerable?" is now equivalent to asking whether $v_{m+1} > 0$.

As we have seen in the preceding sections, features are correlated with vulnerabilities. How can we use this information to answer the above questions? Both questions can

[4] For this, we use χ^2 tests if the entries in the corresponding contingency table are all at least 5, and Fischer exact tests if at least one entry is 4 or less.


Vulnerability vector **v**
Row vector $\mathbf{x}_3$
nsDOMClassInfo has had 10 vulnerability-related bug reports
nsDOMClassInfo imports "nsiXPConnect.h"
Import matrix *X*

		<stdio.h>	"util.h"	"nsStackFrame.h"	<sys/file.h>	"sslimpl.h"	"nsIXPConnect.h"	"orkinEnv.h"	"btree.h"	"nsIJavaDOM.h"	"jvmp.h"
nsCOMArray	0	1	0	0	0	1	0	0	0	1	0
nsIDocument.h	1	0	0	1	0	0	0	1	0	1	1
nspr_md.h	0	0	1	1	0	0	0	0	0	1	0
nsDOMClassInfo	10	0	0	1	0	0	1	0	0	0	0
EmbedGtkTools	0	0	0	0	0	0	0	0	1	0	0
MozillaControl.cpp	0	0	1	0	1	0	0	0	0	0	0

Figure 6: The feature matrix X and the vulnerability vector v for imports. The rows of X contain the imports of a certain component as a binary vector: x_{ik} is 1 if component i imports import k. The vulnerability vector contains the number of vulnerability-related bug reports for that component.

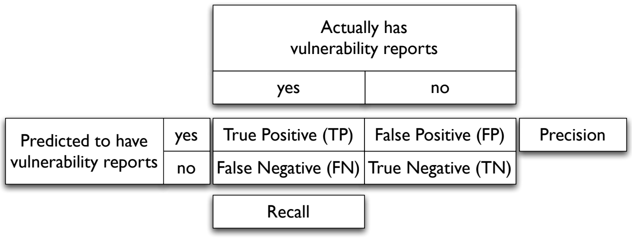
be posed as machine-learning problems. In machine learning, a parameterized function f, called a *model*, is trained using training data X and $\mathbf{y}$, so that we predict $\hat{\mathbf{y}} = f(X)$. The parameters of f are usually chosen such that some measure of difference between $\mathbf{y}$ and $\hat{\mathbf{y}}$ is minimized. The question, "Is this component vulnerable?" is called *classification*, and "Is this component more or less vulnerable than another component?" can be answered with *regression*: by predicting the number of vulnerabilities and then ranking the components accordingly.

In our case, X would be the project's feature matrix, and $\mathbf{y}$ would be the vulnerability vector $\mathbf{v}$. We now train a model and use it to predict for a new component x'. If it classifies x' as vulnerable, this means that x' has features that were associated with vulnerabilities in other components.

For our model f, we chose support vector machines [36] (SVMs) over other models such as k-nearest-neighbors [13, Chapter 13] because they have a number of advantages. For example, when used for classification, SVMs cope well with data that is not linearly separable. They are also much less prone to overfitting than other machine-learning methods.[5]

5.1 Validation Setup

To test how good our particular set of features work as predictors for vulnerabilities, we simply split our feature matrix to train and to assess the model. For this purpose, we randomly select a number of rows from X and the corresponding elements from $\mathbf{v}$—collectively called the *training set*—and use this data to train f. Then we use the left-over rows from X and elements from $\mathbf{y}$—the *validation set*—to predict whether the corresponding components are vulnerable and to compare the computed prediction with what we already know from the bug database. It is usually recommended that the training set be twice as large as the validation set, and we are following that recommendation. We are not using a dedicated test set because we will not be selecting a single model, but will instead be looking at the statistical properties of many models and will thus not tend to underestimate the test error of any single model [13, Chapter 7].



		Actually has vulnerability reports		
		yes	no	
Predicted to have vulnerability reports	yes	True Positive (TP)	False Positive (FP)	Precision
	no	False Negative (FN)	True Negative (TN)	
		Recall		

Figure 7: Precision and recall explained. Precision is $TP/(TP + FP)$; recall is $TP/(TP + FN)$.

One caveat is that the training and validation sets might not contain vulnerable and neutral components in the right proportions. This can happen when there are so few vulnerable components that pure random splitting would produce a great variance in the number of vulnerable components in different splits. We solved this problem by *stratified sampling*, which samples vulnerable and neutral components separately to ensure the proper proportions.

5.2 Evaluating Classification

For classification, we can now compare the predicted values $\hat{\mathbf{v}}$ with the actual values $\mathbf{v}$ and count how many times our prediction was correct. This gives rise to the measures of *precision* and *recall*, as shown in Figure 7:

- The *precision* measures how many of the components predicted as vulnerable actually have shown to be vulnerable. A high precision means a low number of false positives; for our purposes, the predictor is *efficient.*
- The *recall* measures how many of the vulnerable components are actually predicted as such. A high recall means a low number of false negatives; for our purposes, the predictor is *effective.*

In order to assess the quality of our predictions, consider a simple cost model.[6] Assume that we have a "testing budget" of T units. Each component out of m total components is either vulnerable or not vulnerable, but up front we do not know which is which. Let us say there are V vulnerabilities distributed arbitrarily among the m components and that if we spend 1 unit on a component, we determine for sure whether the component is vulnerable or not. In a typical software project, both V and T would be much less than m.

If we fix T, m, and V, and if we have no other information about the components, the optimal strategy for assigning units to components is simply to choose components at random. In this case, the expected return on investment would be TV/m: we test T components at random, and the fraction of vulnerable components is V/m.

Now assume that we have a predictive method with precision p and that we spend our T units only on components that have been flagged as vulnerable by the method. In this case, the expected return on investment is Tp because the

[5] Two sets of n-dimensional points are said to be *linearly separable* if there exists an $n-1$-dimensional hyperplane that separates the two sets. Overfitting occurs when the estimation error in the training data goes down, but the estimation error in the validation data goes up.

[6] This was suggested to us by the anonymous reviewers.

fraction of vulnerable components among the flagged components is p. If $p > V/m$, the predictive method does better than random assignment. In practice, we estimate V by the number of components already known to have vulnerabilities, V', so we will want p to be much larger then V'/m.

5.3 Evaluating Ranking

When we use a regression model, we predict the *number* of vulnerabilities in a component. One standard action based on this prediction would be *allocating quality assurance efforts:* As a manager, we would spend most resources (such as testing, reviewing, etc.) on those components which are the most likely to be vulnerable. With a prediction method that estimates the number of vulnerabilities in a component, we would examine components of $\hat{\mathbf{v}}$ in decreasing order of predicted vulnerabilities.

Usually, the quality of such rankings is evaluated using Spearman's rank correlation coefficient. This is a number between -1 and 1 which says how well the orderings in two vectors agree. Values near 1 mean high correlation (if the values in one vector go up, then so do the values in the other vector), values near 0 mean no correlation, and values near -1 mean negative correlation (if the values in one vector go up, the values in the other vector go down).

However, this measure is inappropriate within the simple cost model from above. Suppose that we can spend T units on testing. In the best possible case, our ranking predicts the actual top T most vulnerable components in the top T slots. The relative order of these components doesn't matter because we will eventually fix all top T components: while high correlation coefficients mean good rankings, and while bad rankings will produce correlation coefficients near 0, the converse is not true.

Instead, we extend our simple cost model as follows. Let $p = (p_1, \ldots, p_m)$ be a permutation of $1, \ldots, m$ such that $\hat{\mathbf{v}}_p = (\hat{v}_{p_1}, \ldots, \hat{v}_{p_m})$ is sorted in descending order (that is, $\hat{v}_{p_j} \geq \hat{v}_{p_k}$ for $1 \leq j < k \leq m$), and let q and $\mathbf{v}_q$ be defined accordingly. When we fix component p_j, we fix v_{p_j} vulnerabilities. Therefore, when we fix the top T predicted components, we fix $F = \sum_{1 \leq j \leq T} v_{p_j}$ vulnerabilities, but with optimal ordering, we could have fixed $F_{\text{opt}} = \sum_{1 \leq j \leq T} v_{q_j}$ vulnerabilities instead. Therefore, we will take the quotient $Q = F/F_{\text{opt}}$ as a quality measure for our ranking. This is the fraction of vulnerabilities that we have caught when we used p instead of the optimal ordering q. It will always be between 0 and 1, and higher values are better.

In a typical situation, where we have $V \ll m$ and T small, a random ranking will almost always have $Q = 0$, so our method will be better than a random strategy if Q is always greater than zero. In order to be useful in practice, we will want Q to be significantly greater than zero, say, greater than 1/2.

6. CASE STUDY: MOZILLA

To evaluate Vulture's predictive power, we applied it to the code base of Mozilla [34]. Mozilla is a large open-source project that has existed since 1998. It is easily the second most commonly used Internet suite (web browser, email reader, and so on) after Internet Explorer and Outlook.

6.1 Data Collection

We examined Mozilla as of January 4, 2007. Vulture mapped vulnerabilities to components, and then created the feature matrices and the vulnerability vector as described in Sections 3.3 and 4.4.

Phase	Time
Downloading and analyzing MFSAs	5 m
Mapping vulnerabilities to components	1 m
Finding includes	0.5 m
Finding function calls	8 m
Creation of SVM, w/classification and regression	0.5 m

Table 3: Approximate running times for Vulture's different phases.

Table 3 reports approximate running times for Vulture's different phases when applied to Mozilla with imports as the features under consideration. Vulture is so fast that we could envision it as part of an IDE giving feedback in real time (see Figure 10 at the end of the paper).

The 10,452 × 9,481 import matrix would take up some 280 MB of disk space if it were written out in full. The sparse representation that we used [15] required only 230 KB of disk space, however. The 10,452 × 93,265 function call matrix took up 2.6 MB of disk space.

For each feature matrix and the vulnerability vectors, we created 40 random splits using stratified sampling. This ensures that vulnerable and neutral components are present in the training and validation sets in the same proportions. The training set had 6,968 entries and was twice as large as the validation set with 3,484 entries; this is the standard proportion for empirical evaluations of this kind. Finally, we assessed these SVMs with the 40 validation sets.

For the statistical calculations, we used the R system [24] and the SVM implementation available for it [9]. It is very easy to make such calculations with R; the size of all R scripts used in Vulture is just about 200 lines. The calculations were done on standard hardware without special memory sizes or processing powers.

6.2 Classification

The SVM used the linear kernel with standard parameters. Figure 8 reports the precision and recall values for the 40 random splits, both for imports and for function calls. For imports, the recall has an average of 0.45 and standard deviation of 0.04, which means that about half of all vulnerable components are correctly classified:

> *Of all vulnerable components,*
> *Vulture flags 45% as vulnerable.*

For function calls, the precision has a mean of 0.70 and a standard deviation of 0.05.

> *Of all components flagged as vulnerable,*
> *70% actually are vulnerable.*
> *Vulture is much better than random selection.*

6.3 Ranking

The SVM used the linear kernel with standard parameters. The coefficient Q that was introduced in Section 5.3 was computed for imports and function calls, for $T = 30$. It is shown in Figure 9, plotted against F_{opt}. For imports, its mean is 0.78 (standard deviation is 0.04), for function calls, it is 0.82 (standard deviation 0.08).

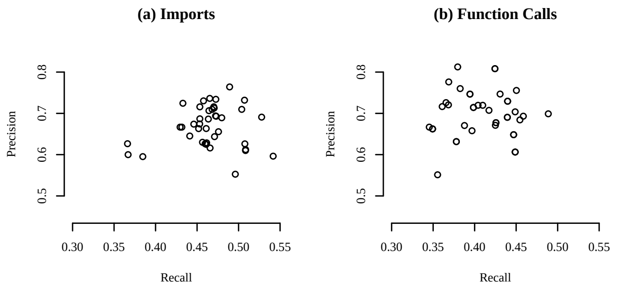


Figure 8: Scatterplot of precision/recall values for the 40 experiments. Figure (a) shows the results for imports, figure (b) shows the results for function calls. The apparent lack of data points is due to overplotting of close-by precision/recall pairs.

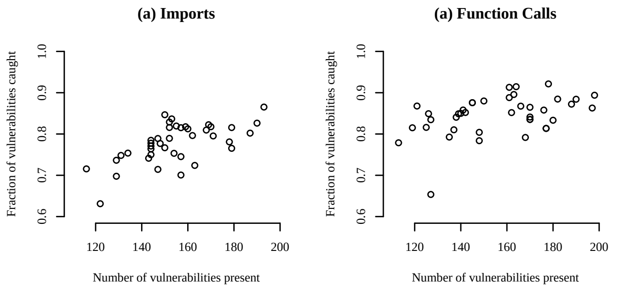


Figure 9: Scatterplot of Q versus F_{opt} for the 40 experiments where $T = 30$. Figure (a) shows the results for imports, figure (b) shows the results for function calls. Higher values are better.

Among the top 30 predicted components, Vulture finds 82% of all vulnerabilities.

Let us illustrate the quality of the ranking by an actual example where $T = 10$. Table 4 shows such a prediction as produced in one of the random splits. Within the validation set, these would be the components to spend extra effort on. Your effort would be well spent, because *all* of the top ten components actually turn out to be vulnerable. (`SgridRowLayout` and `NsHttpTransaction` are outliers, but still vulnerable.) Furthermore, in your choice of ten, you would recall the top four most vulnerable components, two more would still be in the top ten (at predicted ranks 3 and 6), and two more would be in the top twenty (at predicted ranks 7 and 9).

6.4 Discussion

In the simple cost model introduced in Section 5.2, we have $m = 10,452$ and $V' = 424$, giving $V'/m = 0.04$. With $p = 0.65$, we see that Vulture does more than fifteen times better than random assignment.

For ranking, all Q values are higher than 0.6; the average values are way above that. This more than satisfies our criterion from Section 5.3.

Therefore, our case study shows three things. First of all, allocating quality assurance efforts based on a Vulture prediction achieves a reasonable balance between effectiveness and efficiency. Second, it is *effective* because half of all vulnerable components are actually flagged. And third, Vulture is *efficient* because directing quality assurance efforts on flagged components yields a return of 70%—more than two out of three components are hits. Focusing on the top ranked components will give even better results.

Prediction		Validation set	
Rank	Component	BRs	Actual rank
# 1	`NsDOMClassInfo`	10	# 3.5
# 2	`SgridRowLayout`	1	# 95
# 3	`xpcprivate`	7	# 6
# 4	`Jsxml`	11	# 2
# 5	`nsGenericHTMLElement`	6	# 8
# 6	`Jsgc`	10	# 3.5
# 7	`NsJSEnvironment`	4	# 12
# 8	`Jsfun`	15	# 1
# 9	`NsHTMLLabelElement`	3	# 18
# 10	`NsHttpTransaction`	2	# 35

Table 4: The top ten most vulnerable components from a validation set, as predicted by Vulture. The column labeled "BRs" shows the number of vulnerability-related bug reports for that component. Eight of the predicted top ten are actually very vulnerable.

Furthermore, these numbers show that there is empirically an undeniable correlation between imports and function calls on one hand, and vulnerabilities on the other. This correlation can be profitably exploited by tools like Vulture to make predictions that are correct often enough so as to make a difference when allocating testing effort. Vulture has also identified features that very often lead to vulnerabilities when used together and can so point out areas that should perhaps be redesigned in a more secure way.

Best of all, Vulture has done all this automatically, quickly, and without the need to resort to intuition or human expertise. This gives programmers and managers much-needed objective data when it comes to identify (a) where past vulnerabilities were located, (b) other components that are likely to be vulnerable, and (c) effectively allocating quality assurance effort.

7. RELATED WORK

Previous work in this area reduced the number of vulnerabilities or their impact by one of the following methods:

Looking at components' histories. The Vulture tool was inspired by the pilot study by Schröter et al. [29], who first observed that imports correlate with failures. While Schröter et al. examined *general defects,* the present work focuses specifically on *vulnerabilities.* To our knowledge, this is the first work that specifically mines and leverages vulnerability databases to make predictions. Also, our correlation, precision and recall values are higher than theirs, which is why we believe that focusing on vulnerabilities instead of on bugs in general is worthwhile.

Evolution of defect numbers. Both Ozment et al. [23] as well as Li et al. [17] have studied how the numbers of defects and security issues evolve over time. Ozment et al. report a decrease in the rate at which new vulnerabilities are reported, while Li et al. report an increase. Neither of the two approaches allow mapping of vulnerabilities to components or prediction.

Estimating the number of vulnerabilities. Alhazmi et al. use the rate at which vulnerabilities are discovered to build models to predict the number of as yet undiscovered vulnerabilities [2]. They use their approach on entire systems, however, and not on source files. Also, in contrast to Vulture, their predictions depend on a model of *how* vulnerabilities are discovered.

Miller et al. build formulas that estimate the number of defects in software, even when testing reveals no flaws [20]. Their formulas incorporate random testing results, information about the input distribution, and prior assumptions about the probability of failure of the software. However, they do not take into account the software's history—their estimates do not change, no matter how large the history is.

Tofts et al. build simple dynamic models of security flaws by regarding security as a stochastic process [35], but they do not make specific predictions about vulnerable software components. Yin et al. [39] highlight the need for a framework for estimating the security risks in large software systems, but give neither an implementation nor an evaluation.

Testing the binary. By this we mean subjecting the binary executable—not the source code—of the program in question to various forms of testing and analysis (and then reporting any security leaks to the vendor). This is often done with techniques like fuzz testing [19] and fault injection; see the book by Voas and McGraw [38].

Eric Rescorla argues that finding and patching security holes does not lead to an improvement in software quality [25]. But he is talking about finding security holes *by third-party outsiders in the finished product* and not about finding them *by in-house personnel during the development cycle*. Therefore, his conclusions do not contradict our belief that Vulture is a useful tool.

(Statically) examining the source. This is usually done with an eye towards *specific* vulnerabilities, such as buffer overflows. Approaches include linear programming [12], dataflow analysis [14], locating functions near a program's input [8][7], axiomatizing correct pointer usage and then checking against that axiomatization [11], exploiting semantic comments [16], checking path conditions [31], symbolic pointer checking [28], or symbolic bounds checking [26].

Rather than describing the differences between these tools and ours in every case, we we briefly discuss ITS4, developed by Viega et al. [37], and representative of the many other static code scanners. Viega et al.'s requirement was to have a tool that is fast enough to be used as real-time feedback during the development process, and precise enough so that programmers would not ignore it. Since their approach is essentially pattern-based, it will have to be *manually* extended as new patterns emerge. The person extending it will have to have a concept of the vulnerability before it can be condensed into a pattern. Vulture will probably not flag components that contain vulnerabilities that were unknown at training time, but it *will* flag components that contain vulnerabilities that have been fixed before but have no name.

Also, since ITS4 checks local properties, it will be very difficult for it to find security-related defects that arise from the interaction between far-away components, that is, components that are connected through long chains of def-use relations. Additionally, ITS4, as it exists now, will be unable to adapt to programs that for some reason contain a number of pattern-violating but safe practices, because it completely ignores a component's history.

Another approach is to use model checking [3, 4]. In this approach, specific classes of vulnerabilities are formalized and the program model-checked for violations of these formalized properties. The advantage over other formal methods is that if a failure is detected, the model checker comes up with a concrete counter-example that can be used as a regression test case. This too is a useful tool, but like ITS4, it will have to be extended as new formalizations emerge. Some vulnerability types might not even be formalizable.

Vulture also contains static scanners—it detects features by parsing the source code in a very simple manner. However, Vulture's aim is not to declare that certain lines in a program might contain a buffer overflow, but rather to direct testing effort where it is most needed by giving a *probabilistic assessment* of the code's vulnerability.

Hardening the source or runtime environment. This encompasses all measures that are taken to mitigate a program's ability to do damage Hardening a program or the runtime environment is useful when software is already deployed. StackGuard is a method that is representative of the many tools that exist to lower a vulnerability's impact [6]. Others include mandatory access controls as found in AppArmor [5] or SELinux [22]. However, Vulture works on the other side of the deployment divide and tries to direct programmers and managers to pieces of code requiring their attention, in the hope that StackGuard and similar systems will not be needed.

8. CONCLUSIONS AND FUTURE WORK

We have presented empirical evidence that features correlate with vulnerabilities. Based on this empirical evidence, we have introduced Vulture, a new tool that predicts vulnerable components by looking at their features. It is fast and reasonably accurate: it analyzes a project as complex as Mozilla in about half an hour, and correctly identifies half of the vulnerable components. Two thirds of its predictions are correct.

The contributions of the present paper are as follows:

1. A technique for mapping past vulnerabilities by mining and combining vulnerability databases with version archives.

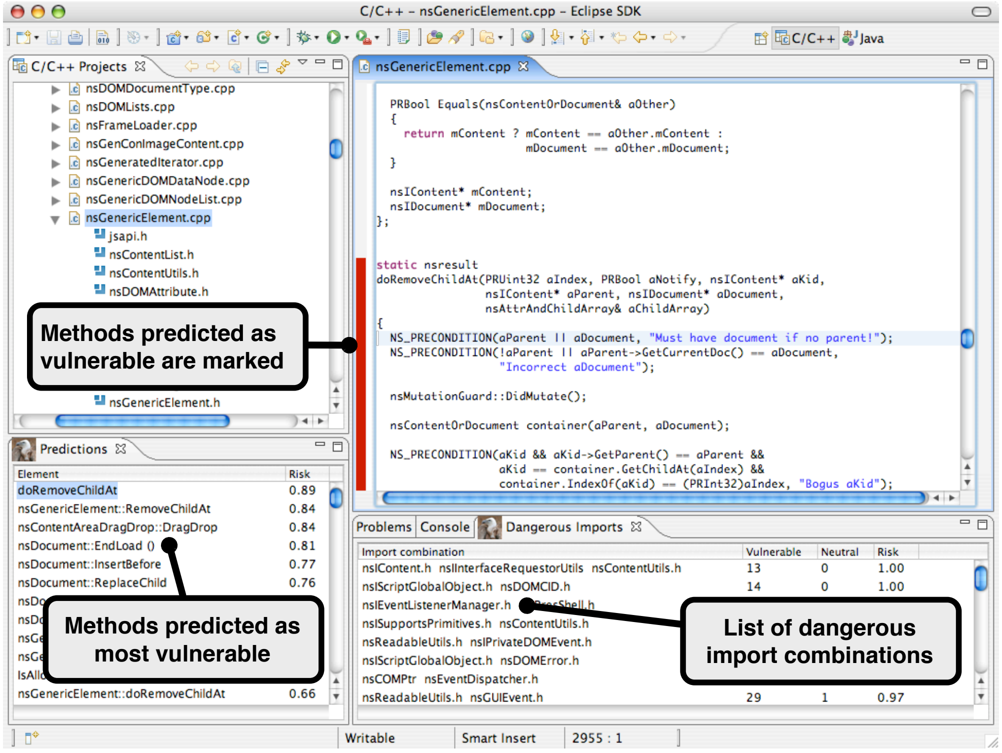
2. Empirical evidence that contradicts popular wisdom saying that vulnerable components will generally have more vulnerabilities in the future.

3. Evidence that features correlate with vulnerabilities.

4. A tool that learns from the locations of past vulnerabilities to predict future ones with reasonable accuracy.

5. An approach for identifying vulnerabilities that automatically adapts to specific projects and products.

[7] The hypothesis of DeCast et al. that vulnerabilities occur more in functions that are close to a program's input is not supported by the present study. Many of Mozilla's vulnerable components, such as `nsGlobalWindow`, lie in the heart of the application.



```
  PRBool Equals(nsContentOrDocument& aOther)
  {
    return mContent ? mContent == aOther.mContent :
                      mDocument == aOther.mDocument;
  }

  nsIContent* mContent;
  nsIDocument* mDocument;
};


static nsresult
doRemoveChildAt(PRUint32 aIndex, PRBool aNotify, nsIContent* aKid,
                nsIContent* aParent, nsIDocument* aDocument,
                nsAttrAndChildArray& aChildArray)
{
  NS_PRECONDITION(aParent || aDocument, "Must have document if no parent!");
  NS_PRECONDITION(!aParent || aParent->GetCurrentDoc() == aDocument,
                  "Incorrect aDocument");

  nsMutationGuard::DidMutate();

  nsContentOrDocument container(aParent, aDocument);

  NS_PRECONDITION(aKid && aKid->GetParent() == aParent &&
                  aKid == container.GetChildAt(aIndex) &&
                  container.IndexOf(aKid) == (PRInt32)aIndex, "Bogus aKid");
```

Predictions

Element	Risk
doRemoveChildAt	0.89
nsGenericElement::RemoveChildAt	0.84
nsContentAreaDragDrop::DragDrop	0.84
nsDocument::EndLoad ()	0.81
nsDocument::InsertBefore	0.77
nsDocument::ReplaceChild	0.76
nsDo	
nsDo	
nsGe	
nsGe	
IsAllo	
nsGenericElement::doRemoveChildAt	0.66

Dangerous Imports

Import combination	Vulnerable	Neutral	Risk
nsIContent.h nsIInterfaceRequestorUtils nsContentUtils.h	13	0	1.00
nsIScriptGlobalObject.h nsDOMCID.h	14	0	1.00
nsIEventListenerManager.h PresShell.h			
nsISupportsPrimitives.h nsContentUtils.h			
nsReadableUtils.h nsIPrivateDOMEvent.h			
nsIScriptGlobalObject.h nsDOMError.h			
nsCOMPtr nsEventDispatcher.h			
nsReadableUtils.h nsGUIEvent.h	29	1	0.97

Figure 10: Sketch of a Vulture integration into Eclipse. Vulture annotates methods predicted as vulnerable with red bars. The view *"Predictions"* lists the methods predicted as most vulnerable. With the view *"Dangerous Imports"*, a developer can explore import combinations that lead to past vulnerabilities.

6. A predictor for vulnerabilities that only needs a set of suitable features, and thus can be applied before the component is fully implemented.

Despite these contributions, we feel that our work has just scratched the surface of what is possible, and of what is needed. Our future work will concentrate on these topics:

Characterizing domains. We have seen that empirically, features are good predictors for vulnerabilities. We believe that this is so because features characterize a component's domain, that is, the type of service that it uses or implements, and it is really the domain that determines a component's vulnerability. We plan to test this hypothesis by studies across multiple systems in similar domains.

Fine-grained approaches. Rather than just examining features at the component level, one may go for more fine-grained approaches, such as caller-callee relationships. Such fine-grained relationships may also allow vulnerability predictions for classes or even methods or functions.

Evolved components. This work primarily applies to predicting vulnerabilities of new components. However, components that already are used in production code come with their own vulnerability history. We expect this history to rank among the best predictors for future vulnerabilities.

Usability. Right now, Vulture is essentially a batch program producing a textual output that can be processed by spreadsheet programs or statistical packages. We plan to integrate Vulture into current development environments, allowing programmers to query for vulnerable components. Such environments could also visualize vulnerabilities by placing indicators next to the entities (Figure 10).

In a recent blog, Bruce Schneier wrote, "If the IT products we purchased were secure out of the box, we wouldn't have to spend billions every year making them secure." [27] One first step to improve security is to learn where and why current software had flaws in the past. Our approach provides essential ground data for this purpose, and allows for effective predictions where software should be secured in the future.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. We also thank the Mozilla team for making their databases available. David Schuler and Andrzej Wasylkowski provided valuable feedback on earlier revisions of this paper. Thomas Zimmermann is funded by a stipend from the DFG-Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

Vulture is part of the "Mining software archives" project at Saarland University. For more information, see

`http://www.st.cs.uni-sb.de/softevo/`

9. REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. 20th Int'l Conf. on Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, September 1994.

[2] Omar Alhazmi, Yashwant Malaiya, and Indrajit Ray. *Security Vulnerabilities in Software Systems: A*

Quantitative Perspective, volume 3645/2005 of *Lecture Notes in Computer Science*, pages 281–294. Springer Verlag, Berlin, Heidelberg, August 2005.

[3] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Proc. 11th Annual Network and Distributed System Security Symposium (NDSS)*, pages 171–185, February 2004.

[4] Hao Chen and David Wagner. MOPS: An infrastructure for examining security properties of software. In *Proc. 9th ACM Conf. on Computer and Communications Security (CCS)*, pages 235–244, November 2002.

[5] Crispin Cowan. Apparmor linux application security. `http://www.novell.com/linux/security/apparmor/`, January 2007.

[6] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conf.*, pages 63–78, San Antonio, Texas, January 1998.

[7] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, June 2005.

[8] Dan DaCosta, Christopher Dahn, Spiros Mancoridis, and Vassilis Prevelakis. Characterizing the security vulnerability likelihood of software functions. In *IEEE Proc. 2003 Int'l Conf. on Software Maintenance (ICSM'03)*, September 2003.

[9] Evgenia Dimitriadou, Kurt Hornik, Friedrich Leisch, David Meyer, and Andreas Weingessel. *e1071: Misc Functions Department of Statistics (e1071), TU Wien*, 2006. R package version 1.5-13.

[10] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *Proc. Int'l Conf. on Software Maintenance (ICSM'03)*, Amsterdam, Netherlands, September 2003. IEEE.

[11] Pascal Fradet, Ronan Caugne, and Daniel Le Métayer. Static detection of pointer errors: An axiomatisation and a checking algorithm. In *European Symposium on Programming*, pages 125–140, 1996.

[12] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *10th ACM Conf. on Computer and Communications Security (CCS)*, October 2003.

[13] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer Series in Statistics. Springer Verlag, 2001.

[14] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy.* May 2006.

[15] Roger Koenker and Pin Ng. *SparseM: Sparse Linear Algebra.* R package version 0.73.

[16] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, pages 177–190, August 2001.

[17] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Proc. Workshop on Architectural and System Support for Improving Software Dependability 2006*, pages 25–33. ACM Press, October 2006.

[18] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In *Knowledge Discovery in Databases: Papers from the 1994 AAAI Workshop*, pages 181–192, 1994.

[19] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study reliability of UNIX utilities. *Communications* , 33(12):32–44, 1990.

[20] K.W. Miller, L.J. Morell, R.E. Noonan, S.K. Park, D.M. Nicol, B.W. Murrill, and M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE Transactions on Software Engineering*, 18(1):33–43, January 1992.

[21] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proc. 29th Int'l Conf. on Software Engineering.* ACM Press, November 2005.

[22] National Security Agency. Security-enhanced linux. `http://www.nsa.gov/selinux/`, January 2007.

[23] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *Proc. 15th Usenix Security Symposium*, pages 93–104, August 2006.

[24] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0.

[25] Eric Rescorla. Is finding security holes a good idea? *IEEE Security and Privacy*, 3(1):14–19, 2005.

[26] Radu Rugina and Martin Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *Proc. ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 182–195. ACM Press, 2000.

[27] Bruce Schneier. Do we really need a security industry? *Wired*, May 2007. `http://www.wired.com/politics/security/commentary/securitymatters/2007/%05/securitymatters_0503`.

[28] Berhard Scholz, Johann Blieberger, and Thomas Fahringer. Symbolic pointer analysis for detecting memory leaks. In *Proc. 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 104–113. ACM Press, 1999.

[29] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proc. 5th Int'l Symposium on Empirical Software Engineering*, pages 18–27, New York, NY, USA, September 2006.

[30] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proc. Second Int'l Workshop on Mining Software Repositories*, pages 24–28, May 2005.

[31] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. In *Proc. 24th Int'l Conf. on Software Engineering*, New York, NY, USA, May 2002. ACM Press.

[32] The Mozilla Foundation. Bugzilla. `http://www.bugzilla.org`, January 2007.

[33] The Mozilla Foundation. Mozilla foundation security advisories. `http://www.mozilla.org/projects/security/known-vulnerabilities.html`, January 2007.

[34] The Mozilla Foundation. Mozilla project website. `http://www.mozilla.org/`, January 2007.

[35] Chris Tofts and Brian Monahan. Towards an analytic model of security flaws. Technical Report 2004-224, HP Trusted Systems Laboratory, Bristol, UK, December 2004.

[36] Vladimir Naumovich Vapnik. *The Nature of Statistical Learning Theory.* Springer Verlag, Berlin, 1995.

[37] John Viega, J. T. Bloch, Tadayoshi Kohno, and Gary McGraw. Token-based scanning of source code for security problems. *ACM Transaction on Information and System Security*, 5(3):238–261, 2002.

[38] Jeffrey Voas and Gary McGraw. *Software Fault Injection: Innoculating Programs Against Errors.* John Wiley & Sons, 1997.

[39] Jian Yin, Chunqiang Tang, Xiaolan Zhang, and Michael McIntosh. On estimating the security risks of composite software services. In *Proc. PASSWORD Workshop*, June 2006.