Compile-time Detection of Machine Image Sniping

Martin Kellogg
Paul G. Allen School of Computer Science and Engineering
University of Washington
Seattle, USA
kelloggm@cs.washington.edu

***Abstract*—Machine image sniping is a difficult-to-detect security vulnerability in cloud computing code. When programmatically initializing a machine, a developer specifies a machine image (operating system and file system). The developer should restrict the search to only those machine images which their organization controls: otherwise, an attacker can insert a similarly-named malicious image into the public database, where it might be selected instead of the image the developer intended. We present a lightweight type and effect system that detects requests to a cloud provider that are vulnerable to an image sniping attack, or proves that no vulnerable request exists in a codebase. We prototyped our type system for Java programs that initialize Amazon Web Services machines, and evaluated it on more than 500 codebases, detecting 14 vulnerable requests with only 3 false positives.**

***Index Terms*—pluggable types, AMI sniping, AWS, EC2, Java, lightweight verification, DescribeImagesRequest**

I. The Problem

A client of a cloud services provider can create virtual computers programmatically. An *image* is the virtual computer's file system; it includes an operating system and additional installed software, and so it determines what code runs on the virtual computer.

For example, a client of Amazon Web Services indicates what image to use via the DescribeImagesRequest API (like the client in fig. 2). This API (which is shown in fig. 1) suffers a serious security vulnerability [1].

There are three safe ways to select which image to use when sending a request to the API:

- Use the withImageIds method to specify a globally unique image ID.
- Use the withOwners method to restrict the images searched to those owned by the requester or another trusted party.
- Use the withFilters method to set criteria that restrict the image to one that is owned by a trusted party using the "owner", "owner-id", "owner-alias", or "image-id" filters.

The unsafe example in fig. 2 uses the "name" filter without an owner filter, which causes the API to return all the images that match the name. This enables the so-called "AMI (Amazon Machine Image) sniping attack" [1], in which an attacker intentionally creates a new image whose name collides with the desired image, permitting the attacker to surreptitiously inject their own code onto newly allocated machines. Any call that searches the public database without specifying some information that an adversary cannot fake is vulnerable to a sniping attack and should be forbidden.

```
package com.amazonaws.services.ec2.model;

class DescribeImagesRequest {
 DescribeImagesRequest() {...}
 withOwners(String... owners) {...}
 withFilters(Filter... filters) {...}
 withImageIds(String... imageIds) {...}
}
```

Fig. 1: The DescribeImagesRequest **API. A client constructs a** DescribeImagesRequest**, modifies it via the** with* **methods, then sends it to AWS to obtain a machine image.**

```
request = new DescribeImagesRequest();
filter = new Filter("name", "RHEL-7.5_HVM_GA");
request.withFilters(filter);
api.describeImages(request);
```

Fig. 2: Client use of the DescribeImagesRequest API that is vulnerable to an AMI sniping attack.

II. Our approach

Our key insight is that **only certain combinations of method calls should be permitted**. We therefore designed a type system that records, for each object, the set of methods that have been invoked on that object. This information is represented in a *type qualifier* named @CalledMethods. A type qualifier is a modifier on a type, which makes it more specific. Our implementation operates on Java programs, which represent type qualifiers via annotations, which start with @.

@CalledMethods(*methodList*) Object obj means that methods in *methodList* have *definitely* been called on obj. For example, if a() and b() have been called on obj, then obj has type @CalledMethods({"a", "b"}). As additional methods are called using the same receiver expression, the type of that expression is refined to include more methods. Figure 3 shows part of the type hierarchy. The subtyping rule is:

$$\frac{A \supseteq B}{\text{@CalledMethods(A)} \sqsubseteq \text{@CalledMethods(B)}}$$

Our type system also supports a disjunctive type, @CalledMethodsPredicate(P), which permits specifications like "withOwners $\vee$ withImageIds". A type qualified by @CalledMethods(A) is a subtype of a type qualified by @CalledMethodsPredicate(P) if and only if A makes P evaluate to true.

To detect and prevent erroneous calls, an API designer writes a specification—that is, they write types on formal parameters. For the AWS AMI sniping example, the corresponding specification is written on the parameter to the describeImages API in the AWS SDK (for presentation, the full specification has been shortened):

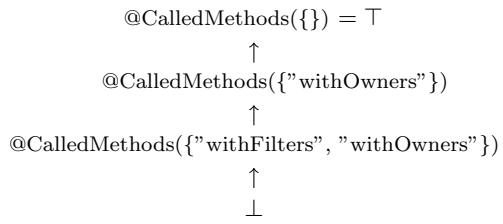

Fig. 3: Part of the type hierarchy for representing which methods have been called. If an expression's type has qualifier @CalledMethods("withFilters", "withOwners")**, then the methods "**withFilters**" and "**withOwners**" have definitely been called on the expression's value. Arrows represent subtyping relationships. The diagram shows a part of the type hierarchy; the full hierarchy is a lattice of arbitrary size.**

TABLE I: The results of applying our AMI detection tool to two corpora of closed- and open-source projects. We measured non-comment, non-blank lines of Java code.

	Open-source	Closed-source
Projects	39	509
Lines of code	490K	8.7M
Annotations	4	29
True Positives	1	13
False Positives	2	1

```
DescribeImageResponse describeImages(
   @CalledMethodsPredicate("withImageIds || withOwners")
   DescribeImageRequest request);
```

Given this specification for describeImages, the typechecker rejects any call whose receiver has not had either withImageIds or withOwners called on it. This specification is sound: it prevents all AMI sniping attacks.

In general, the typechecker either:

- proves that a codebase only contains calls that are consistent with the specification, or
- issues an error, indicating possibly-defective code.

We implemented this type system for Java using the Checker Framework [2]. The framework provides local type inference, among other conveniences, which means that programmers only need write type qualifiers on formal parameters, fields, and return types. However, because most types are inferred, even these are rarely needed.

III. Evaluation

We evaluated our approach on two corpora of codebases:

- 39 open-source codebases from GitHub. We searched GitHub for projects that use the describeImages API and then filtered out (for technical reasons) projects whose root directory did not contain a Gradle or Maven build file and those that did not build with a Java 8 compiler. We also discarded duplicates (copies and forks) and the AWS Java SDK itself.
- 509 codebases from an anonymous industrial partner that contain one or more calls to the describeImages() API.

Table I shows the results. The tool overall achieved 82% precision, and required one annotation per 278,000 lines of code. It is easy to find the places where annotations are needed, because the tool initially issues warnings in those locations.

```
request = new DescribeImagesRequest();
if (imageIds != null) {
   request.setImageIds(Arrays.asList(imageIds));
}
result = ec2Client.describeImages(request);
```

Fig. 4: A true positive AMI sniping vulnerability in an API in Netflix's SimianArmy project.

The true positive we discovered in the open-source evaluation was in the project Netflix/SimianArmy; the relevant code appears in fig. 4. If the list of image ids is null, then the code (by design) fetches every AMI available. Though the method's documentation does not say so, it is incumbent on any caller of this code to filter the result after the fact.

Both false positives in the open-source experiments were due to imprecision in the Checker Framework's generic inference algorithm, and were easily discarded as spurious (since they occurred in code that manifestly did not call the AWS API). We reported the imprecision to the framework's maintainers, who acknowledged it as a bug. When that bug is fixed, our tool will achieve 100% precision on those codebases.

IV. Related Work

No other static analysis exists which detects an image sniping attack. AWS acknowledged the vulnerability, but cannot revoke this widely-used API nor change its behavior incompatibly. Their proposed mitigation is "to follow the best practice and specify an owner" [3]. An independent security researcher published instructions to detect if running virtual machines are impacted, but said that following best practices is the best mitigation [4]. A sound static analysis like ours does not depend on programmers to remember to use the best practice.

There are static analyses that detect security issues, including Coverity [5], CogniCrypt [6] and CryptoGuard [7]. None of these tools contains rules for finding image sniping attacks.

Our type system can be viewed as a limited form of typestate [8] in which objects can only accumulate method calls. This limited form can be efficiently implemented without an expensive, potentially imprecise alias analysis.

V. Conclusion and Future Work

We have presented an effective, sound analysis for machine image sniping. Our analysis for method calls is applicable to other problems beyond machine image sniping. For example, our system can model object construction via the builder and dependency injection patterns, permitting us to guarantee correct construction at compile time rather than suffering a run-time error. In fact, the machine image sniping problem described here is an example of a builder object (the DescribeImagesRequest) that can be constructed in a dangerous way, resulting in a silent security vulnerability.

Acknowledgments

Thanks to my collaborators Manu Sridharan, Manli Ran, and Michael D. Ernst.

References

[1] "CVE-2018-15869," Available from MITRE, CVE-ID CVE-2018-15869, 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-15869

[2] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for Java," in *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, Seattle, WA, USA, July 2008, pp. 201–212.

[3] J. Bicha and N. Alvine, "Cve-2018-15869: –owners flag isn't mandatory," https://github.com/aws/aws-cli/issues/3629, 2018, accessed 5 June 2019.

[4] S. Piper, "Investigating malicious amis," https://summitroute.com/blog/2018/09/24/investigating_malicious_amis/, 2018, accessed 5 June 2019.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[6] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," in *European Conference on Object-Oriented Programming (ECOOP)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[7] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, Danfeng, Yao, and M. Kantarcioglu, "Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects," 2018.

[8] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, no. 1, pp. 157–171, 1986.