The dark side of the code

Olgierd Pieczul[1,2] and Simon N. Foley[2]

[1] Ireland Lab, IBM Software Group, Dublin, Ireland.
olgierdp@ie.ibm.com
[2] Department of Computer Science, University College Cork, Ireland.
s.foley@cs.ucc.ie

Abstract. The literature is rife with examples of attackers exploiting unexpected system behaviours that arise from program bugs. This problem is particularly widespread in contemporary application programs, owing to the complexity of their many interconnected parts. We consider this problem, and consider how runtime verification could be used to check an executing program against a model of expected behaviour generated during unit testing.

"Beware of bugs in the above code; I have only proved it correct, not tried it."
—Donald Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion, 1977

1 Introduction

Contemporary application systems are implemented using an assortment of high-level programming languages, software frameworks and third party components. While this may help lower development time and cost, the result is a complex system of interoperating parts whose behaviour is difficult to fully and properly comprehend. This difficulty of comprehension often manifests itself in the form of program coding errors that are not directly related to security requirements but can have a significant impact on the security of the system [16]. For example, while an application may enforce the correct access controls, its programmer may have mistakenly relied on the software development framework providing particular code injection defenses; alternatively, the framework developer may have mistakenly relied on its users implementing their own injection defenses, or, simply, that nobody had anticipated and/or understood the injection vulnerability. In a study of developers by Oliveira et. al. [23], it was found that 53% of its participants knew about a particular coding vulnerability, however they did not correlate it with their own programming activity unless it was explicitly highlighted. It is, therefore, not surprising that all of the OWASP Top 10 security risks [26] relate to implementation flaws, with the majority in the form of common coding mistakes. Two security vulnerabilities that received wide media coverage in 2014, Heartbleed [2] and Shellshock [1] are further examples of such mistakes.

Given the complexity of contemporary applications, and the manner of their development, we argue that there will always be some aspect of their behaviour (ranging from application level to low-level system calls) that a programmer may not have fully considered or comprehended. We refer to this as the dark side of the code; a *security gap* that can exist between the expected behaviour and the actual behaviour of the code. Improper or incomplete comprehension means that security controls may not have been considered for the security gap and, as a consequence, the unexpected behaviour arising from the code may give rise to a security vulnerability. One might argue that encapsulation and programming by contract [21] could eliminate these security gaps; or, that one might attempt to model all unexpected behaviours in the security gap in terms of a Dolev-Yao style attacker [13, 30] and verify that the application code is in turn robust to failure against this attacker. However, these approaches still require a full and proper comprehension of system components and their interoperation (in terms of formal specification) which, in itself, can have security gaps, regardless of the challenge of scaling these techniques to contemporary application systems.

In this paper we explore the dark side of the code and show how security gaps can emerge in applications. We argue that the security gap can be reduced by using anomaly detection style techniques to monitor the expected behaviour, against the actual behaviour, of individual applications and components. Our preliminary experimental results suggest that this approach can be used to identify a variety of security vulnerabilities that arise from programming error.

2 Contemporary application development

Current software frameworks enable developers to focus on the high-level functionality of the application by hiding low-level detail. System infrastructure detail such as DBMS, local file systems and memory is encapsulated as object storage; network connectivity is abstracted in terms of remote resource access, and user interaction and presentation is supported via a range of standard interfaces. In this paper we use a running example to explore the coding of a contemporary application. Despite being simple, the example is sufficient to illustrate some of the challenges in using these frameworks and to identify some of the unexpected vulnerabilities in the security gap.

Consider a web application that provides an online facility for users to manage and organize website bookmarks that is synchronised across different user devices. The application provides a web interface and REST API, supporting website snapshot images and metadata, browsing, searching and tracking bookmark use. The application can be built with little effort using contemporary tools, frameworks and libraries. For example, Listing 1 provides the code implementing bookmark creation request. The application is hosted on a web application server that handles network connection along with HTTP request. We assume that the application uses a web MVC framework to parse request parameters `address` and bookmark `title` and calls `addBookmark`. The framework is also responsible for exception interception, error rendering and providing re-

sponses to the client. Inside the method `addBookmark`, a utility library ([4]) is

```
1  void addBookmark(String address, String title) throws
       Exception {
2      Image small = WebUtils.snapshot(address, 160, 120);
3      Image large = WebUtils.snapshot(address, 1200, 800);
4      Bookmark b = new Bookmark(address, title, small,
           large);
5      DataStore.save(b);
6  }
```

Fig. 1. Web application - bookmark creation

used to render website snapshot images (lines 2–3) and a persistence framework (such as Hibernate [3]) to save the bookmark in a relational database (line 5).

While the high-level application code is clear and easy to follow, the program abstractions that are used mean that the typical programmer will not overly concern themselves with the specifics of the low-level behaviour of the underlying framework infrastructure. For example, at Line 5 the application uses the persistence framework to save a bookmark. The developer expects that the framework will make a connection to a database (or reuse an existing one), formulate an SQL statement from the bookmark object fields and execute it. Similarly, creation of website snapshot images (lines 2–3) is handled using a single call to an external library. The documentation, such as javadoc provided with the `WebUtils.snapshot()` source code in Listing 2, provides limited information about the method behaviour. ¿From this, the developer can learn that it accesses the website specified by the URL, renders it and returns an image of specified dimensions. The programmer can expect that the library will verify the correctness of the provided address (as an exception is thrown for an "incorrect URL"), and that it will check for "communication problems" while the website is being accessed.

Studying the source code of `WebUtils.snapshot()` method in Listing 2, we can see that the library, used by the application, is also implemented at a similarly high level of abstraction. All of the logic related to accessing the remote website in order to create the snapshot is covered by lines 13–15 using the Java Platform API `URL` and `URLConnection` classes. Looking at the first lines of its documentation [24], the developer learns that a *"Class `URL` represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web"*. The documentation informs its reader that in the case of a malformed URL, the constructor will throw an exception, which they decide to forward to the consumer to handle. Furthermore, the documentation [25] specifies that `URL.openConnection()` method returns *"a connection to the remote object referred to by the URL"*. And that `URLConnection.getInputStream()` *"returns an input stream that reads from this open connection"*. The documentation also

```
1  /**
2  * Create an image snapshot for a website
3  * @see    #render(InputStream)
4  *
5  * @param  website URL address of the website
6  * @param  w       image width
7  * @param  h       image height
8  * @return         Image containing website snapshot
9  * @throws IOException communication problem
10 * @throws MalformedURLException incorrect URL
11 */
12 static public Image snapshot(String website, int w, int
       h) throws IOException, MalformedURLException {
13     URL url = new URL(website);
14     URLConnection connection = url.openConnection();
15     InputStream input = connection.getInputStream();
16     Image image = render(input, w, h);
17     return image;
18 }
```

Fig. 2. Library method

states that a `SocketTimeoutException` is thrown *"if the read timeout expires before data is available for read"*.

3 Securing what is understood

The convenience of using abstractions and their ability to handle security threats relieves the developer from having to consider much of the low-level details. For example, because object persistence frameworks do not require construction of SQL queries, the programmer need not consider sanitizing user input with respect to the SQL language. Similarly, letting the MVC framework provide the Web presentation layer can reduce programmer concerns about application output interfering with the output context, such as HTML, XML and JSON. This does not excuse the programmer from considering security issues entirely, rather the emphasis is on the security controls that are relevant to the application code.

Regardless of the effectiveness of the programming abstractions, it is reasonable to expect that the developer does understand some of the underlying system operation in order to identify possible threats and to counter them with adequate security controls. For example, although not directly referenced in the application code, it may be anticipated that the application will communicate over HTTP with the remote website in order to create a snapshot. Thus, the application should be permitted to make HTTP connections that are, to some degree, controlled by application users through the URLs they enter, and this may be a security threat. In this case, the application could be used to access

systems—in the local network where it is hosted—that are not normally accessible from the Internet. A malicious user may, by adding a bookmark to the URL in the local network, such as `http://10.0.0.1/router/admin`, attempt to access systems that he should not have access to. In order to address this threat, the developer can code a security control in the application that verifies that the URL's host does not point to a local IP address, before calling the library to create a snapshot. For example, the `addBookmark()` method can begin with:

```
// [...]
InetAddress addr =
    InetAddress.getByName(url.getHost()));
if (addr.isSiteLocalAddress())
     throw new SecurityException();
// [...]
```

4 The security gap

To a casual reader, the bookmark application (or even the `WebUtils` library) code does not openly perform TCP/IP operations. The above threat was identified based on the programmer's expectation of low-level application behaviour. Correlating high level application behaviour (accessing URLs) with the threat (user-controlled network traffic) is a human task and, as such, is prone to human error. Failure to implement adequate security controls may not necessarily mean that the developers are unaware of the threat or neglect security. As observed previously, despite understanding a security vulnerability, a developer may unwittingly write code containing the vulnerability [23]. The cognitive effort that is required to anticipate security problems is much greater if the details are abstracted.

Consider again the bookmark application extended with the security control to prevent local URL access. Despite appearances, an attacker can bypass this check as follows. We first note that the HTTP protocol [12] allows a server to redirect the client to another URL in order to fulfill the request through a defined status code (such as 302) and a header.

- An attacker sets up a website that redirects to the local target machine and adds a bookmark to that website.
- The attacker's website (public) URL will be accepted as not local and action `WebUtils.snapshot()` is called.
- The Java library will access the website and in `url.openConnection()` in the implementation of `snapshot` effectively follow the URL, effectively connecting to a local address.

In order to prevent this attack it is necessary for the programmer to modify the utility library to explicitly handle redirects and verify the IP address each time, before accessing the URL. This approach, however, may suffer a TOCTTOU vulnerability. In this case, there is a time gap between the verification

of the IP address and the HTTP connection to the corresponding URL. Within that time gap, the mapping between host name and IP address may be modified. While past responses will typically be cached by the resolver, the attacker may prevent the caching by creating a record with lowest possible Time To Live value supported by Domain Name System [22], that is, 1 second. This is a variant of a DNS Rebinding attack [17].

Perhaps, and rather than trying to implement the network-related security controls in the application, a better strategy is to consider this a matter for system network configuration. In this case, it should be the systems and network administrators, not the developers, who need to handle the problem by implementing adequate firewall rules. While transferring administrative burdens to the consumer is a common practice [11], it also pushes the abstraction further and may make the threat equally difficult to identify.

Regardless of how this network protection is implemented, the web application still contains an even more serious and unexpected vulnerability. It allows application clients to access custom files from the web server's file system. The root cause is the fact that RFC3986 [9] specifies that a Uniform Resource Identifier can be `file:`, in addition to `http:` and `https:`. In this case,

- an application client, may create a bookmark for the address URL such as `file:///etc/passwd` and,
- the application generates an image representing file contents.

This behaviour may not have been anticipated by the application developer who, upon reading the documentation, understood that `WebUtils.snapshot()` should be called with a *"website address"* and which throws an exception if that address is *"incorrect"*. Similarly, the library developer might have been misled by the Java `URL`/`URLConnection` documentation and method names referring to *"connections"* and *"sockets"* and did not expect that their code could be used to access regular files. While, the `URL` class javadoc [24] includes a reference to `file:` URL scheme, it appears only once, in one of the constructors' documentation. Other platforms include similar, often misleading, features. For example the function `file_get_contents` in PHP, despite its name, allows accessing remote resources if the URL is provided as a file name [27].

To avoid this vulnerability, the developer must implement specific code that checks whether the URL specifies a website address. However, other URL-related problems may emerge. For example, in another part of the application, it may be required to verify whether a URL matches a list of accepted URLs. The Java `equals()` method can be used (explicitly or implicitly via the `List.contains()` method) as a standard way to test object equality. Using this method is convenient when comparing URL objects, as it respects that a host name and protocol are case insensitive and some port numbers are optional. For example, `http://example.com/`, `http://example.com:80/` and `HTTP://EXAMPLE.COM/` are equal URLs despite their different string representation. What may not be anticipated by the programmer, is that when comparing two URL objects, the method resolves their host names and considers them equal if they point to the same IP address. In this case `http://example.com/` is considered *equal* to

`http://attacker.com/`, provided that the attacker has targeted their host to `example.com`'s IP address. This unexpected behaviour may lead to security vulnerabilities if URLs are for used for white/black listing, or to ensure the Same Origin Policy [8]. While the behaviour of `URL.equals()` is documented and the corresponding security issues are considered [10], a developer may not consider checking that part of documentation to be necessary, especially if the code may not explicitly invoke the method.

This example demonstrates how programming oversights, in what seemed to be trivial, high-level application code, can result in series of security issues whose identification and prevention requires an in-depth understanding of low-level libraries and a number of network protocols. In today's systems of interoperating components, the security gap between the expected behaviour and the actual behaviour is unavoidable. Pushing the responsibility to understand everything onto developers is expecting them to be omniscient, is not realistic, and is in effect, security theatre.

5 Verifying expectation

An application system program has a security gap when a developer's misunderstanding means that an attacker can exploit the difference between its expected behaviour versus its actual behaviour and, for which security controls do not exist. In this section we outline our ongoing investigation on how the gap can be reduced by checking the runtime behaviour of the component against a predefined model of its expected behaviour. Runtime verification [7] is the process of observing system execution and validating that specified properties are upheld, or that the execution is consistent with a testing oracle. For example, a predicate stating that each `acquire` has a matching `release` in a (re-entrant) Lock class [19]. However, based on our earlier observations, we argue that a typical developer would still be unable to capture all properties about expected behaviour; a security gap remains. The challenge is to construct a model of expected behaviour that helps to reduce the security gap.

We use data mining techniques to infer patterns of expected behaviour from execution traces that are generated during the testing phase of the development lifecycle. Such system trace mining techniques have been used elsewhere to infer acceptable behaviour/policies for anomaly detection [14,28] process mining [5,6], security policy mining [15, 20] and fault detection [18]. Expected behaviour is not exactly the same as the *normal* behaviour upon which anomaly detection is typically based. Normal behaviour is system-specific, and relates to system configuration, infrastructure, usage patterns and so forth. Typically, it is established for a particular instance of the system based on monitoring its operation under normal circumstances. The expected behaviour corresponds to the anticipated behaviour of the system under all expected circumstances. It represents all of the activity that the system is capable of performing. Some activity may be expected, recognized as possible in the system in general, but not considered normal in a particular system instance.

An approach for inferring behavioural models from system logs was proposed in the seminal work of Forrest at al. [14]. System behaviour is modeled in terms of a set of *n-grams* of system call operations present in the system log. As the system executes, its operations are compared against this model of 'normal' and, if the sequence does not match known n-grams, it may be considered anomalous. This approach is limited to identifying short-range correlations between operations in traces and can miss interesting behaviours by not considering transactional behaviour [29]. In our work we use use the richer behavioural norms [28, 29] which can identify repeating patterns of parameterized behaviour from the system trace.

Intuitively, behavioural norms are sequences of parametrized actions. For example, one of the norms that represent the actions that result from accessing an `http:` URL is the following sequence of method invocations:

```
net.url("get","http",$1,$2);
socket.connect($1,$3);
socket.resolve($1);
socket.connect($4,$3);
socket.read();
...;
socket.close()
```

The norm represents a repeating pattern of behaviour discovered in a system trace. It is not generated by static analysis of the code. Action parameter attributes are identified as static values (such as `"get"`) or free variables (such as `$1`) that can be bound to some value when matching an instance of the norm against runtime behaviour. In mining a system trace, our analysis algorithm [28] has proven quite effective in identifying the action attributes that remain static versus those that can change, and how they correlate, each time the norm is matched. For example, the norm above is the outcome of a `get` on an `http:` URL to some host `$1`, resulting in a socket `connect` and `resolve` on involving the same host and subsequent `connect` on a resolved IP address `$1`.

We carried out a preliminary evaluation of the use of behavioural norms as a means of identifying coding errors in the security gap. A set of conventional unit tests were run against the `WebUtils` library, testing both positive and negative cases. For example, the website image snapshot was tested with a number of URLs and expected error conditions, such as incorrect URL, unresolvable host, interrupted connection, and so forth. Note that none of the tests attempted to open `file:` URLs, as such behaviour was not expected as possible. The Java build process for the library was extended to capture a trace during test execution. This trace is a sequence of the low-level actions that result from testing `WebUtils`, and contain not just those actions from the snapshot library, but also all others, such URL methods, etc. This trace of all method invocations was analyzed for behavioural norms and the resulting model of expected behaviour was included in the library manifest. A Java aspect was used to provide runtime verification using the Java security manager. Note that neither generating

the behavioural model nor its runtime verification requires any change to the original library code and is applicable to any Java component.

The web application was redeployed with its new library and tested. While the original positive and negative test cases were still effective (testing expected behaviour, such as creating images from websites) we found that the unexpected behaviour, such as creating images using local files, was no longer possible. In this case, the behavioural norm model permitted snapshot transactions that involved `http:` URLs, but identified a snapshot transaction using `file:` URLs as anomalous. Note that the generated model does not prevent the application from carrying out all `file:` URL operations, just those that occur within a snapshot transaction. In practice, other parts of the bookmark web application may be expected to access files, and general file accessing behaviour is part of the expected behaviour of the application as a whole.

We are currently investigating how this approach can identify other coding errors that lead to security vulnerabilities. In preliminary experiments, we have observed that a number of typical programming errors, including Insecure Direct Object Reference, TOCTTOU, access control flaws and broken authentication manifest themselves through distinguishably different low-level behaviour and, as such, could be detected using that technique.

6 Conclusion

Given the complexity of contemporary applications, we argue that there will always be a security gap between the code's actual behaviour and the behaviour expected by the programmer. The cognitive overload on the programmer increases with the level of the programming abstractions used and increases the likelihood of errors that lead to security vulnerabilities. We propose taking a runtime verification style approach to check an executing program against a model of expected behaviour that is generated during unit testing. While our initial experiments provide some evidence that this approach has potential, we note that its effectiveness depends greatly on the completeness of the unit testing at exercising expected behaviour.

Acknowledgement. This work was supported, in part, by Science Foundation Ireland under grant SFI/12/RC/2289 and the Irish Centre for Cloud Computing and Commerce, an Irish national Technology Centre funded by Enterprise Ireland and the Irish Industrial Development Authority.

References

1. Bash code injection vulnerability via specially crafted environment variables, `https://access.redhat.com/articles/1200223`
2. The heartbleed bug, `http://heartbleed.com/`
3. Hibernate, `http://hibernate.org`
4. The Spring framework. `https://spring.io`

5. Accorsi, R., Stocker, T.: Automated privacy audits based on pruning of log data. In: EDOCW. pp. 175–182 (2008)
6. Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. In: Proceedings of the 6th International Conference on Extending Database Technology: Advances in Database Technology. pp. 469–483. EDBT '98, Springer-Verlag (1998), http://dl.acm.org/citation.cfm?id=645338.650397
7. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Verification, Model Checking, and Abstract Interpretation. pp. 44–57. Springer (2004)
8. Barth, A.: The Web Origin Concept. Request For Comments 6454, Internet Engineering Task Force (Dec 2011), http://www.ietf.org/rfc/rfc6454.txt
9. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax. Request for Comments 3986, Internet Engineering Task Force (Jan 2005), http://www.ietf.org/rfc/rfc3986.txt
10. Carnegie Mellon University: CERT Secure Coding Standards – VOID 2 MET21-J. Do not invoke equals() or hashCode() on URLs, https://www.securecoding.cert.org/confluence/x/5wHEAw
11. Davis, D.: Compliance defects in public-key cryptography. In: Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6. pp. 17–17. SSYM'96, USENIX Association, Berkeley, CA, USA (1996), http://dl.acm.org/citation.cfm?id=1267569.1267586
12. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. Request For Comments 2616, Internet Engineering Task Force (Jun 1999), http://www.ietf.org/rfc/rfc2616.txt
13. Foley, S.: A non-functional approach to system integrity. IEEE Journal on Selected Areas in Communications 21(1) (Jan 2003)
14. Forrest, S., Hofmeyr, S., Somayaji, A., Longstaff, T.: A sense of self for unix processes. In: IEEE Symposium on Security and Privacy. pp. 120–128 (1996)
15. Frank, M., Buhmann, J., Basin, D.: On the definition of role mining. In: Proceedings of the 15th ACM Symposium on Access Control Models and Technologies. pp. 35–44. SACMAT '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1809842.1809851
16. Gollmann, D.: Software security – the dangers of abstraction. In: The Future of Identity in the Information Society, IFIP Advances in Information and Communication Technology, vol. 298, pp. 1–12. Springer Berlin Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-03315-5_1
17. Jackson, C., Barth, A., Bortz, A., Shao, W., Boneh, D.: Protecting browsers from DNS rebinding attacks. In: In Proceedings of ACM CCS 07 (2007), http:///crypto.stanford.edu/dns/dns-rebinding.pdf
18. Jiang, G., Chen, H., Ungureanu, C., Yoshihira, K.: Multi-resolution abnormal trace detection using varied-length n-grams and automata. In: Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on. pp. 111–122 (June 2005)
19. Jin, D., Meredith, P.O., Lee, C., Roşu, G.: Javamop: Efficient parametric runtime monitoring framework. In: Proceedings of the 34th International Conference on Software Engineering. pp. 1427–1430. ICSE '12, IEEE Press, Piscataway, NJ, USA (2012), http://dl.acm.org/citation.cfm?id=2337223.2337436
20. Kuhlmann, M., Shohat, D., Schimpf, G.: Role mining - revealing business roles for security administration using data mining technology. In: Proceedings of the

Eighth ACM Symposium on Access Control Models and Technologies. pp. 179–186. SACMAT '03, ACM, New York, NY, USA (2003), http://doi.acm.org/10.1145/775412.775435
21. Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992), http://doi.ieeecomputersociety.org/10.1109/2.161279
22. Mockapetris, P.: Domain names - concepts and facilities. Request for Comments 1034, Internet Engineering Task Force (Nov 1987), http://www.ietf.org/rfc/rfc1034.txt
23. Oliveira, D., Rosenthal, M., Morin, N., Yeh, K.C., Cappos, J., Zhuang, Y.: It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In: Proceedings of the 30th Annual Computer Security Applications Conference. pp. 296–305. ACSAC '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2664243.2664254
24. Oracle: Java Platform API Specification – URL (2014), http://docs.oracle.com/javase/7/docs/api/java/net/URL.html
25. Oracle: Java Platform API Specification – URLConnection (2014), http://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html
26. OWASP Foundation: OWASP Top 10 2013, https://www.owasp.org/index.php/Top_10_2013
27. The PHP Group: PHP Manual – file_get_contents, http://php.net/manual/en/function.file-get-contents.php
28. Pieczul, O., Foley, S.: Discovering emergent norms in security logs. In: Communications and Network Security (CNS - SafeConfig), 2013 IEEE Conference on. pp. 438–445 (2013)
29. Pieczul, O., Foley, S.: Collaborating as normal: detecting systemic anomalies in your partner. In: Security Protocols XXII. Lecture Notes in Computer Science, vol. 8809. Springer Berlin Heidelberg (2014)
30. Ryan, P.: Mathematical models of computer security. In: Focardi, R., Gorrieri, R. (eds.) Foundations of Security Analysis and Design, Lecture Notes in Computer Science, vol. 2171, pp. 1–62. Springer (2001)