УДК 004.056:004.651.5
DOI: 10.31673/2409-7292.2024.030010

Suprunenko I. O., Rudnytskyi V. M.

DYNAMIC MESSAGE VARIANT IN ADAPTIVE LOGGING METHOD

Observability is an essential part of software systems. As the scale and outreach of modern technologies grow, it becomes increasingly important to be able to pinpoint and diagnose software issues in a timely manner. One approach that is commonly used by developers involves utilizing a technique called “logging”, most likely in a form that is based on the idea of outputting log messages coupled with level of severity. This combination helps to group and categorize different reporting messages for processing afterwards. But sometimes this is not enough as some applications might be so complex and sophisticated that severity-only categorization does not scale properly. To solve this issue an adaptive logging method was introduced. It adds the concept of “log tags” and a special configuration that allows to include or exclude specific tags or their combinations. This article takes the idea of adaptive logging a step further and introduces a new plane of adaptability with dynamic message variants: a specific type of log messages with the ability to override reporting information “on the fly” without changing the source code. At first the motivation and necessity for such functionality is described in abstract terms, then a formalized model of proposed change is presented. A detailed explanation and reasoning behind certain data structures that make dynamic messages possible is presented, providing a reasonable amount of architectural considerations to make implementation in different environments and programming languages more achievable. At the end special attention is paid to some important aspects and requirements that should be carefully considered by implementers when writing their own version of adaptive logging method. The results of applying the proposed update to adaptive logging method would allow developers to have even more tools to extract information about system’s execution and incorrect behaviors easier and with greater detail.

Keywords: information security, cyber threats, observability, adaptive logging, dynamic execution.

Introduction

In the modern landscape of software development, the aspect of information security is hardly one that can be easily neglected. With software systems growing in scale and complexity, storing big amounts of personalized information, private data files, sensitive corporate or government information, it is increasingly more important to provide a sufficient level of protection against different threats coming from both malicious actors as well as from erroneous and incorrect usage from the inside.

Literature review

One of the latest worldwide challenges that the society faced on an unprecedented scale was the global COVID-19 pandemic. Many people were forced to change their lifestyles, their habits and even the way they work. But together with that different cybersecurity threats also changed. It is estimated that from February to March 2020 the number of recorded spam emails turned out to be 220 times higher compared to previous period, with other most common cybersecurity threats during similar period being Distributed Denial of Service (DDOS) attacks, ransomware, mobile threats, as well as malicious websites and domains (Khan *et al*., 2020). It was also reported that malicious scammers took advantage of the situation, hacking people connected to laptops, computers, tablets and phones, stealing sensitive data, using that stolen data to withdraw money from people’s accounts or to run loan scams. As a result, the reported number of frauds in 2020 was 42% higher than in 2019, because cybercriminals took advantage of the fact that many physical stores were unable to continue working as usual and moved to online business models (Alawida, et al., 2022). Naturally there was a shift in the structure of information security domain related to pandemic: network privacy became a trending topic, which seems to be related to work-from-home models and transition to cloud infrastructure by the industry, while sub-domains such as device skimmers or credit card breaches became less popular (Kumar, et al., 2022).

The solution to the issues with privacy appeared in a form of Virtual Private Networks (VPN). In Poland, for example, according to data that was recorded right at the beginning of first lockdowns, one of the consequences of the measures the government took to prevent spread of the COVID 19 was that usage of Open VPN software product almost doubled (Karpowicz, 2021). Some challenges, however, cannot be fixed by just one new technology and require more thorough analysis and assessment. Studies have shown that remote control and assistance of staff would become more

important because of the need to address the bottlenecks in providing secure ways for workers to work remotely, at the same time technologies such as artificial intelligence and machine learning must be included as a part of threat detection capabilities to identify and respond to adverse behaviours faster and without human intervention (Baz, et al., 2021).

A more recent challenge was presented in the form of a global Windows-based machines outage caused by an update of Crowdstrike's antivirus product. It is already estimated that the cost of this disruption might be as high as $4-$6 billion dollars. Some key takeaways presented by Dr. George Shaji (When Trust Fail…, 2024) are: the interdependence between software vendors has to be stress tested to reveal overdependence risks, a proper financial cyber risk quantification is essential to have an adequate level of investment into resilience and response capabilities, trusted domestic alternatives and open standards need to be properly funded to battle supply risks in domains like antiviruses, microprocessors or cloud providers. Interestingly enough this incident presented a different view on the information security concerns, meaning that instead of being the result of a coordinated and successful attack, the root cause seems to be more of a failure in development-release process of critical software, emphasizing that in addition to proper protection of confidentiality, integrity and availability of information, it is also essential to keep a check on the systems that process that information, making sure that those have sufficient degree of quality assurance and observability.

A typical solution that is used in order to make a software system more transparent is to utilize logging, which can be described as a practice of outputting some information about the system's state in a form of textual messages usually coupled with severity level that adds more details about the importance of a particular invocation. This practice has accompanied software development for quite some time and it even has different related cloud based solutions, such as CloudWatch service in Amazon Web Services (AWS), that helps to monitor resources, applications and infrastructure in the cloud by collecting different data such as metrics, events and log-data, configuring automated responses or actions that happen if some limits have been exceeded (Routavaara, 2020). Even though the basic form of logging is pretty well-formed and well-known, the topic is still being researched by scientists. One such recent development is presented as a SCLogger contextualized logging statement generation approach (Li, et al., 2024), that aims to solve several limitations of logging statements, created using generative models, such as limited static scope of those statements, inconsistent style between different output values and missing type information of the variables that are being logged. The resulting performance is said to be higher than from other large language models and the context-aware prompt structure incorporates static domain knowledge, improving automatic log statement generation.

One of the authors' previous work introduces an improvement over the common severity-based logging approach in the form of an "adaptive logging method" (Suprunenko and Rudnytskyi, 2024). It adds an additional layer of configuration to well-known "log message and severity level" structure (presented as log tags), which allows to apply more precise filtering logic, narrowing the search area and limiting the amount of data that needs to be processed in order to find the issue in a software system. With a proper message passing mechanism in place it becomes possible to launch this reconfiguration without losing the runtime state of an observed program. However, one limitation of the presented method is that it basically becomes just another component of the software product that logging is being applied to, which in turn leads to considerably tighter coupling which might be unwanted in some use cases. This paper looks into augmenting adaptive logging method in such a way that would make this connection less restrictive and more configurable.

The subject of study is the observability aspect of information security.

The object of research is the adaptive logging method as a strategy of improved observability in software systems.

The objective of this research is to develop a mechanism that would allow adaptive logging method to change the contents of log messages in a manner similar to changing executed log invocations based on configured tags. This would aid greatly in development and testing

environments allowing developers to have deeper insight not only into a specific part of a system, but also into various internal details of runtime such as class instances, variables, methods, etc.

Materials and methods

Main methods used in this work are abstraction and formalization. At first the value of dynamic reporting capabilities is described in an abstract and isolated form without any direct relation to specific platforms, applications or software programs. Then all the necessary parameters are formalized and presented as equations and structures, sometimes augmenting ones that were introduced in the basic adaptive logging mechanism.

Results and discussion

In its basic form adaptive logging method adds an improvement over common severity-only based reporting by introducing log tags and configuration object, that allow to filter out certain log invocations, leaving only those that are needed at the moment. This ability, while being more flexible than general approach, only allows to remove or append information from predefined and preformatted log call sites. This might work for environments in which source code does not change that often, like production deploys, but during development or staging (basically, “close to real life” testing) phases it sometimes is necessary to be able to do more. A reasonably complex application often consists of different modules - pieces of software created by the current development team or some other third party, that help to prototype faster and develop modular software with some parts being written and tested by many other developers, increasing the quality of that particular method or class. Such programs are often distributed and installed using prebuilt artifacts that might be very different from code in the current development environment and don’t have any means of simple manipulation and tweaking if something is not right. In some cases, it is possible to look through the code of this outer dependency and figure out what the issue is because its code has relatively minor changes compared to the original. But it is also possible to have modules built with fairly complex compilation and packaging pipelines, and so looking into those becomes complicated. Even tools such as interactive debuggers might be of no use if the code being inspected does not have all the necessary parts (e.g. source maps, which are files that map generated textual data to original program file). In such cases being able to reliably output human readable information about code execution, altering it when necessary, can help a lot while searching for issues during the investigation phase.

The signature for adaptive logging method’s log invocation (1) looks as follows:

$$f_{log\ adp} = f(Sev, M, T_{incl}), \quad (1)$$

where *Sev* – a severity of current log invocation; *M* – a message (most likely a string of text); T_{incl} – a set of tags that describe current invocation. Main mechanism for altering the reporting details works through filtering based on desired tags, which are compared to the ones related to a particular log invocation. With this the developer is able to say not only when they want to log something based on severity, but also if a particular tag, or group of tags, should be displayed, while others are skipped. While this mechanism is by design non-intrusive and does not alter the program in any meaningful way (only reporting about its execution), the next step would be to improve the notion of log message M in order to add new layer of adaptability.

The desired functionality is proposed to look like this: message M should exist in two forms that are easily distinguishable and encode two possible behaviours - either output a string literal which is formed during development phase (with possible interpolation of some runtime values, but those are statically linked to the resulting message and there is no way to change them afterwards) or run some computation based on specific parameters accessible from the call site and output the result (this behaviour should use easily stringifiable representation to allow for processing in many platforms and programming languages as not all of them might reliably work with runtime structures such as functions, classes, instances, recursive structures, etc.). The new equation (2) for adaptive logging method now looks like this:

$$f_{log\ adp} = f(Sev, M_{DU}, T_{incl}), \tag{2}$$

where the newly introduced parameter M_{DU} is a representation of two possibilities described above. The "DU" part in its index stands for "discriminated union" (the name is taken from TypeScript programming language (Rehman, 2023)), which is another name for an algebraic data type "sum" (Algebraic data type…, 2024) and it emphasizes the requirement of having a straightforward way to distinguish between "text string" or "dynamic execution" types of log invocations. This is achieved through the usage of a common "discriminator" property and in the case of adaptive logging method is proposed to look as follows:

$$M_{DU} = \begin{cases} \begin{cases} type: "static" \\ msg: string \end{cases} \\ \begin{cases} type: "dynamic" \\ params: List < string > \\ bodyId: string \end{cases} \end{cases} \tag{3}$$

The first half of the union is identified using "static" literal value of a common "discriminator" property ("type") and message is represented as a "string" data type (sequence of characters, text, etc.). It's important to note that this formal definition of static message does not mention the interpolation of runtime values and it is purposefully that way as from the perspective of adaptive logging method implementation there is no need to know or understand how a given log message was formed, only its contents matter.

The second half is what allows for dynamic reporting and its "discriminator" property value equals string literal "dynamic". Typically, in programming languages behaviours are encoded with reusable chunks of code called functions. The same idea applies here, but in a more serializable and dynamic format: "params" and "bodyId" properties together form a definition of a function, where "params" is a list of names of parameters that will be passed to this function and "bodyId" is a string reference to the textual representation of a source code for a given adaptive logging implementation. Important to note that "bodyId" should just be a unique enough identifier that would allow one to locate and retrieve a particular function body from elsewhere (more on this later). Described approach is heavily influenced by both "eval" global function and "Function" constructor in JavaScript programming language (JavaScript: the first…, 2020), as well as by "exec" built-in in Python programming language (Built-in Functions — Python…, 2024).

To properly construct and execute this new dynamic message variation function bodies have to be retrieved using "bodyId". Technically it is possible to write bodies right in log invocations and that would be a step further, compared to plain string in "static" message type, but this does not give the required level of adaptability as it is still a source code written somewhere inside the program (just as a string, not parsed and not executed) and changing it would require to rebuild the module again. The reference at log function call sites must be as stable as possible. The same issue appears with parameters that will be passed to this dynamic method, but the strategy that can be used there is to capture as many variables from local execution scope as possible and pass all of them, allowing the code from the function body to extract only those values it requires. Adaptive logging method initialization and reintialization procedures are both based on init function (4) with following definition:

$$f_{init} = f(Sev, C) \tag{4}$$

where *Sev* – the lowest severity level runtime is expected to report; C – special configuration object that establishes a mapping between tags and "include" or "exclude" actions, configuring what log invocations should be executed.

Addition of "dynamic" message variant requires a new configuration setting (5) that would bind "bodyId"-s and their corresponding contents:

$$M_{dyn} = Map < string, string > \quad (5)$$

where "Map" represents a "hash map"-like structure that helps to easily retrieve body (value) based on "bodyId" (key). And because (4) is that exact special mechanism that allows to configure adaptive logging method implementation during runtime, it is a very natural place to add this hash map as the bodies of dynamic functions for new message variant can be easily overridden in exactly the same manner as log tags or severity. There is one important consideration regarding this: it is highly advisable for an arbitrary adaptive logging implementation to have Map structure created using mechanisms of a particular programming language that allow for easy serialization/deserialization procedures as that presents more opportunities for implementation of initialization method using local file based configuration objects or several other communication approaches like Unix process signals or Unix domain sockets.

With two main components of dynamic messages for an adaptive logging method described (new message variant signature and additional configuration parameter), it is necessary to point out some special parts that require additional attention. First of all, there exists a possibility that for unknown reason "bodyId" in a log invocation fails to map to any actual function body in hash map. While this is generally an unexpected behaviour and should not happen in properly designed systems, the implementation has to be ready for such occurrence. As logging in general can be seen more as a helper functionality around the actually valuable code, the solution should have as little impact on the program as possible. And so if picking between two possibilities of either failing loud and clear in order to spot the error as early as possible, or silently skipping particular log trace, it is advised to implement the latter. This will be more or less evident in development or testing scenarios as developers would expect to see some reporting when there is none, and if it somehow ends up in a production environment – this is a noop. Secondly, it is necessary to design for possible execution failure of the dynamic method and process it accordingly. Because the code passed through text-based communication means to the initialization function is not necessarily well-formed and error-less, it is advised to properly catch possible errors and yet again fall back to silent ignore rather than bringing down the whole program altogether (generally for the same reasons as described with unmatched "bodyId"). A more advanced processing might include a validation procedure of both dynamic body and the return value that could check if the return value matches some expected shape and if the body is a valid source code which uses only allowed functions, statements and other capabilities of a programming language it is implemented in. A more detailed view into this topic is a bit too complex for current discussion and should probably be addressed in a dedicated work. And finally, as was mentioned before, the idea is heavily influenced by specifics of modern programming languages such as JavaScript and Python, and so it might limit the potential implementation targets for adaptive logging method with dynamic messages. It seems that this issue does not have a proper generalized solution and as such is considered to be a responsibility of an implementer of adaptive logging method. The basic form with string-only messages should still be usable in many compiled environments as it is based on simple branching logic.

Conclusions

Observability in software systems plays a significant role and should not be easily neglected. This paper takes a deeper look into adaptive logging method as an improvement over the common severity-only approach. Based on use cases, such as consumption of highly transformed (minified or even compiled) source code in reusable third party or internal modules, a new requirement is defined: it is necessary to be able to properly investigate parts of the software system that could be completely unreadable by human and cannot be reliably investigated because of some apparent limitations in interactive tools such as debuggers. The basic form of adaptive logging method only offers a small

degree of adaptability in this regard as when applied with a static message shape it conceptually becomes just another dependency of a program, and any change to it might require a considerable rebuild process. Instead an alternative message variant is introduced, forming a discriminated union with the original one to allow for ease of disambiguation. This “dynamic” variant offers a much higher level of flexibility based on utilization of stable unique identifiers and configurable “hash map”-like structure resulting in a logging capability that is similar to different methods of runtime code evaluation in some interpreted programming languages. The override functionality for this map structure naturally fits the general reinitialization logic that is already present in a form of log tags and severity update capability in adaptive logging method. The further research could potentially focus on possible solutions to the source code validation issue, caused by serialized representation of function bodies, providing more context and information that would serve as a basis for more deliberate error processing decisions compared to current “fail silently if something goes wrong” strategy.

References

1. Khan, N.A., Brohi, S.N., & Zaman, N. (2020). Ten deadly cyber security threats amid Covid-19 pandemic. TechRxiv, 1-7. DOI: 10.36227/techrxiv.12278792.v1.
2. Alawida M., Omolara A.E., Abiodun O.I., Al-Rajab A. (2022). A deeper look into cybersecurity issues in the wake of Covid-19: A survey. Journal of King Saud University - Computer and Information Sciences,Volume 34, Issue 10, Part A. Pages 8176-8206, ISSN 1319-1578. https://doi.org/10.1016/j.jksuci.2022.08.003.
3. Kumar R., Sharma S., Vachhani C., Yadav N. What changed in the cyber-security after COVID-19? Computers & Security, Volume 120, 102821, ISSN 0167-4048. https://doi.org/10.1016/j.cose.2022.102821.
4. Karpowicz, M.P. (2021). Covid-19 pandemic and internet traffic in Poland: Evidence from selected regional networks. Journal of Telecommunications and Information Technology, 3, 86-91. DOI: 10.26636/jtit.2021.154721.
5. Baz, M., Alhakami, H., Agrawal, A., Baz, A., Khan, R.A. (2021). Impact of COVID-19 pandemic: A cybersecurity perspective. Intelligent Automation & Soft Computing, 27(3), 641-652. https://doi.org/10.32604/iasc.2021.015845.
6. Dr. A. Shaji George. (2024). When Trust Fails: Examining Systemic Risk in the Digital Economy from the 2024 CrowdStrike Outage. Partners Universal Multidisciplinary Research Journal, 1(2), 134–152. https://doi.org/10.5281/zenodo.12828222.
7. Routavaara I. (2020). Security monitoring in AWS public cloud. Bachelor’s Thesis. Technology Information and Communication Technology. JAMK University of Applied Sciences.
8. Li Y., Huo Y., Zhong R., Jiang Z., Liu J., Huang J., Gu J., He P., Lyu M.R. (2024). Go Static: Contextualized Logging Statement Generation. ACM International Conference on the Foundations of Software Engineering. https://doi.org/10.48550/arXiv.2402.12958.
9. Suprunenko, I., & Rudnytskyi, V. (2024). On specifics of adaptive logging method implementation. Bulletin of Cherkasy State Technological University, 29(1), 36-42. https://doi.org/10.62660/bcstu/1.2024.36.
10. Rehman B. (2023). A Blend of Intersection Types and Union Types. Abstract of thesis for the degree of Doctor of Philosophy at The University of Hong Kong.
11. Algebraic data type – HaskellWiki. Retrieved from https://wiki.haskell.org/Algebraic_data_type (last accessed at 03 of August, 2024).
12. Wirfs-Brock A, Eich B. (2020). JavaScript: the first 20 years. Proc. ACM Program. Lang. 4, HOPL, Article 77, 189 pages. https://doi.org/10.1145/3386327.
13. Built-in Functions — Python 3.12.4 documentation. Retrieved from https://docs.python.org/3/library/functions.html#exec (last accessed at 04 of August, 2024).

Надійшла 04.08.2024