ISSN 2219-9365

https://doi.org/10.31891/2219-9365-2024-80-5
UDC 004.49

SUPRUNENKO Illia
Cherkasy State Technological University
https://orcid.org/0000-0002-1188-4804
i.o.suprunenko.asp22@chdtu.edu.ua
RUDNYTSKYI Volodymyr
Cherkasy State Technological University
https://orcid.org/0000-0003-3473-7433

VALIDATION OF DYNAMIC MESSAGE VARIANT IN ADAPTIVE LOGGING METHOD

Technology, and software technology in particular, is one of the main drivers of progress in human society. It allows us to solve existing tasks more efficiently and find solutions for problems that were previously too complex to deal with. The outreach and importance of its influence on everyday life is hard to overstate. To be able to satisfy ever-growing demand software solutions become more complex and sophisticated, reaching millions users and solving numerous day to day tasks. But as a result new challenges appear, especially related to cybersecurity. And it is not only about integrity, availability and confidentiality aspects, but also – equally important – about control and observability. As the scale of software systems grows more and more, tracking their state and behavior becomes more challenging.

This research takes a closer look at observability aspect of cybersecurity, in particular at adaptive logging method for software systems and its dynamic message variant which introduced the ability to have dynamic computations executed before outputting observational information about the system. The flexibility it introduces also makes possible for undesirable side effects to occur as the dynamic nature of such messages allows for execution of virtually any valid code samples. The main purpose of this work is to demonstrate a solution that would allow validation and prevent unwanted code execution. To achieve this an approach that uses analysis of abstract syntax trees with JSON-based validation logic is demonstrated. The natural tree-like structure of ASTs, as well as close resemblance between generated representation and JSON-schemas, allows for precise and easily configurable processing that reduces the possibility of unexpected behaviors. The integration into an existing formal basis of adaptive logging method is also demonstrated with provided justification for required additions and their specifics.

Keywords: cybersecurity, observability, logging, validation, abstract syntax trees, JSON-schema.

СУПРУНЕНКО Ілля, РУДНИЦЬКИЙ Володимир
Черкаський державний технологічний університет

ВАЛІДАЦІЯ ДИНАМІЧНИХ ПОВІДОМЛЕНЬ МЕТОДУ АДАПТИВНОГО ЛОГУВАННЯ

Технології, та технології програмного забезпечення, є одним із головних рушіїв прогресу в людському суспільстві. Вони дозволяють вирішувати існуючі задачі більш ефективно, а також знаходити рішення для проблем, які були занадто складними для розв'язання. Охоплення та важливість їх впливу на повсякденне життя важко переоцінити. Щоб мати змогу задовольнити зростаючий попит, програмні рішення стають більш складними, вирішуючи численні повсякденні задачі для мільйонів користувачів. Однак як наслідок з'являються нові виклики, в тому числі пов'язані з інформаційною безпекою. І мова не лише про аспекти цілісності, доступності та конфіденційності, але також - що не менш важливо - про контроль та спостережність. Із зростанням масштабів систем програмного забезпечення, все більшим викликом є задача спостереження за їх станом та поведінкою.

Ця робота розглядає аспект інформаційної безпеки, що стосується спостережності, а саме метод адаптивного логування для систем програмного забезпечення та динамічну варіацію лог-повідомлень, що дозволяє виконувати динамічні обчислення перед виводом інформації про систему. Окрім більшої гнучкості дана варіація також робить можливими небажані сторонні ефекти, оскільки динамічна природа дозволяє виконання майже будь-якого коду. Мета цієї роботи полягає в демонстрації рішення, що дозволить перевірити та запобігти виконанню небажаного коду. Для цього використано підхід, що використовує аналіз абстрактних синтаксичних дерев на основі JSON-схем. Деревовидна структура АСД, а також подібність між згенерованою структурою та JSON-схемами дозволяє точніше та простіше налаштовувати обробку динамічного коду, що в свою чергу зменшує ймовірність несподіваної поведінки. Також продемонстровано інтеграцію в існуючу формальну основу методу адаптивного логування з поясненнями необхідних змін та їх особливостей.

Ключові слова: кібербезпека, спостережуваність, журналювання, валідація, абстрактні синтаксичні дерева, JSON-схема.

INTRODUCTION

Globalized modern world is heavily reliant on software technologies. Reading news online, checking bank accounts, searching information, learning, shopping and other activities have been digitized to a considerable degree so that it might even be quite odd to do some of those differently. The ease of global network access that handheld devices provide only ensures that the Internet is not going to disappear any time soon. But this convenience comes at a cost and not only the performance and user-friendliness of such systems is in high demand – cybersecurity considerations are also essential.

LITERATURE REVIEW

Ease of access to different services that the Internet provides can sometimes feel as a fundamental undeniable right, but it is very well possible even for absolutely harmless, beneficial and publicly open technologies to become a victim of malicious actors. In October 2024 the biggest organization that deals with storing historical Internet data – Internet Archive (and their Wayback Machine service) – was attacked and kept in unusable state for several days ("Internet Archive Services Update: 2024-10-17", 2024). Together with a distributed denial of service (DDOS) attack, an exposure of email addresses and encrypted passwords occurred. Even though the crawling services were said to be back online, the damage had already been done. A BlackMeta hacktivist group claimed to be behind this attack and promised to conduct new attacks.

Potential threats can be presented in a much more subtle manner than direct and plain DDOS attacks. And even then the number of affected people can be quite high. In September 2024 a vulnerability was recorded in the National Vulnerability Database ("NVD – CVE-2024-9680", 2024) that described a successful attack resulting in a code execution in the content process after exploiting "use-after-free" related software bug. It affected a considerable range of Firefox web browser releases and originated from Animation timelines, which is a module related to Cascading Style Sheets and is generally not expected to cause such critical issues. As software becomes more complex and more rich in features – more surface for attack gets exposed. Physical threats to software systems are just as dangerous as ones that are entirely virtual and can even affect even those systems that were specifically designed to be much less susceptible to external threats. For example, there exists a possibility of an attack conducted on or air-gapped systems that are physically separated from the Internet and other networks (Guri, 2024). It was shown that it is possible to emit radio signals from a compromised computer and make it emit data, such as files, images, password, biometric information, that can be easily captured even by the off-the-shelf antenna. As a result, the established safety perimeter with protection mainly based on separation from other computers might turn out to be helpless against an attack from a new and unexpected angle.

The pursuit of new and better technology, new technological wonders that should make everyday lives easier can also bring about new dangers and cause security concerns. With the rise of "artificial intelligence"-like solutions based on large language models, giants like Microsoft started putting effort into adding those into their products. One such addition, AI-powered Recall Feature for Copilot+ PCs, is said to raise some security concerns among general public, which in turn caused Microsoft to delay the rollout of this controversial solution ("Microsoft Delays AI-Powered Recall Feature for Copilot+ PCs Amid Security Concerns", 2024). And following plans included first releasing to a smaller subset of users to make sure that the experience for end users will be secure and trusted. So it can be stated that security related concerns are still pertinent and should not be easily ignored.

Aside from three fundamental pillars that are integrity, availability and confidentiality, the aspect of observability is just as important. If a software system does not have a sufficient degree of observability, the consequences can be severe. In summer of 2024 a widespread disruption to computers using Microsoft Windows operating systems happened (Ogundipe et al. 2024). The cause was concluded to be a faulty update of Falcon cybersecurity software and affected millions of Windows devices and while not malicious in its intent, it demonstrated (among other things such as the need for progressive release strategies for software products) how dangerous an insufficient degree of certainty in a piece of software can be. The oversights in both management and internal control over what gets shipped and whether it is developed correctly caused a lot of issues for institutions such as Bank of America, the Commonwealth Bank of Australia and London Stock Exchange, which experienced delays in displaying the opening trades (Gudimetla, 2024).

Recent technological advancements, specifically the development of large language models, opened up new ways of working with tasks such as natural language processing, content creation and even decision making in different systems. Despite all of the improvements that it brought to those fields an issue of hallucination in LLMs remains a key challenge (Cleti and Jano, 2024). As those models can sometimes generate content that is factually incorrect, inconsistent or entirely fabricated (yet seems plausible), it is essential to properly debug and refine the internal architecture to prevent such errors from causing harm. The detection methods presented by the authors include named entity recognition, probability-based approaches, as well as prompt engineering and grounding techniques.

There are solutions to the problem of insufficient observability such as monitoring and logging. Those are pretty much widespread and known to the general public and there are lots of available options to use in custom systems, however only big corporations with high expertise (Netflix, Facebook, Google) are generally able to develop appropriate solutions for large scales (Tamburri et al., 2020), while other companies and solutions use the composition of thousands of existing monitoring tools. And it is possible that for some use cases plain monitoring and logging approaches are not enough.

RESULTS AND DISCUSSION

To solve some observability related issues an approach called "adaptive logging method" (Suprunenko & Rudnytskyi, "On specifics of…", 2024) can be used. It is based on a concept of software logging, but with special information (called "log tags") added to each invocation that allows to be more specific. Then, based on a special

configuration description, a programmer can filter out information entries, excluding generation of those, that have no value for the current task at hand. A further improvement of the initial idea, that focused mainly on filtering capabilities combined with re-initialization procedures, is presented in "Dynamic message variant in adaptive logging method" (Suprunenko & Rudnytskyi, 2024). In addition to common and typical text-based messages, another approach is presented that allows to run some processing logic inside an active runtime environment and to inspect different parts of an execution context without introducing major changes to the current state of the codebase. This capability is based on the script-like nature of some languages (such as Python or Javascript) which to some extent limits the portability of updated design of base adaptive logging method, but is required to give even more flexibility and make the development process more transparent. Because of this, the rest of this research is mainly focused on implementations that allow for dynamic code evaluation and in fact takes Javascript programming language as the basis for further development.

As described in "Dynamic message variant in adaptive logging method", there are 2 main equations that describe updated adaptive logging method.

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

which describes a signature of log invocation function, where:

Sev – severity of a given log message, usually represented as a literal string from a predefined set of values, such as "error", "warning", "debug" (particular values are not enforced by the method itself and can be specified by an implementer with a constraint that those should have strict ordering from less important to more important),

M_{DU} – message description which can be presented in one of two forms: either a text-based message that is predefined to some extent, or a script-like "dynamic evaluation" based reference with captured arguments that allows to compute debug values and investigate behaviour of a system "on the fly",

T_{incl} – a set of tags for current invocation.

$$f_{init} = f(Sev, C) \quad (2),$$

which defines the shape for initialization function for an adaptive logging method implementation, where:

Sev – severity, but as a configuration flag for the entire implementation, having a meaning of "only log messages with severity greater than or equal to this one are of interest",

C – configuration object, that consists of settings such as a map of dynamic message bodies to be evaluated and executed with provided parameters, a set of special "and" and "or" segments that fine-tune the desired level of precision a programmer wants to get from a piece of software.

The addition of code that can be assembled and evaluated during runtime execution is a very powerful tool, but the scale of flexibility it introduces is shadowed by some of the flaws of such design. If malformed and incorrect code samples can safely be caught by runtime, there is no built-in way to properly sanitize what exactly gets evaluated. The application of the latest design of adaptive logging method in production environments is a somewhat debatable topic, but for development or staging environments it might serve as a valuable addition. And for those cases it seems beneficial to have the ability to limit (or at least validate) what code is being evaluated on the fly.

Because dynamic messages are represented as textual strings of code, there are several ways to process those, for example using regular expression oriented parsing or utilizing abstract syntax trees (ASTs). As described in "Dynamic source code processing approaches in context of adaptive logging method" (Suprunenko & Rudnytskyi, 2024), AST-based processing provides better capabilities mainly because this is exactly what software parsers are designed to do (break strings of code into sensible structures) and because the resulting representation is easily traversable and can be meaningfully reasoned about.

To properly proceed further it is required to set some constraints and limitations. As mentioned previously, this research is based on an environment for a particular programming language – Javascript (and its platform for writing backend code – NodeJS). As such the examples of code and AST representations are written in it. In order to show how processing of dynamic messages in adaptive logging method can be performed, the reasoning and assumptions are based around a particular use case from real life that would benefit from adaptive logging method, in particular – logging details about http requests in a web server. A popular solution used to write web servers in NodeJs is a framework called "Express.js" ("Express - Node.js web application framework"). At its core it allows to create a web server as a series of routes or routers that listen on a particular HTTP resource path using a specific HTTP method. Each request handler has access to two specific entities: "request" and "response", where "request" serves as a holder for all information and methods related to client HTTP request and "response" is used to form and send HTTP responses. Figure 1 shows an example route on a path "/authorization" using POST method and corresponding output of "request" method in console display.

ISSN 2219-9365

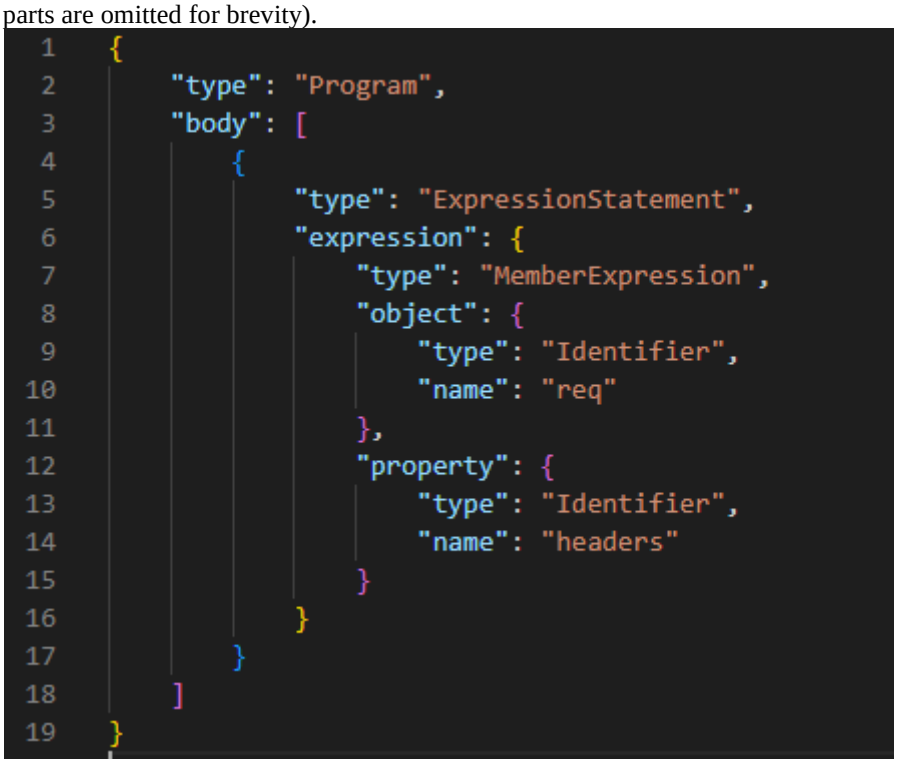
```
13    app.post(
14      '/authorization',
15      (req, res) => {
16        console.log(req);
17        // process request and form
18        // response somehow
19        // ...
20        res.end();
21      }
22    );
<ref *2> IncomingMessage {
  _events: {
    close: undefined,
    error: undefined,
    data: undefined,
    end: undefined,
    readable: undefined
  },
  _readableState: ReadableState {
    highWaterMark: 16384,
    buffer: [],
    bufferIndex: 0,
    length: 0,
    pipes: [],
    awaitDrainWriters: null,
    [Symbol(kState)]: 1315084
  },
  _maxListeners: undefined,
  socket: <ref *1> Socket {
    connecting: false,
    _hadError: false,
    _parent: null,
    _host: null,
    _closeAfterHandlingError: false,
    _events: {
      close: [Array],
      error: [Function: socketOnError],
      prefinish: undefined,
      finish: undefined,
      drain: [Function: bound socketOnDrain],
      data: [Function: bound socketOnData],
      end: [Array],
```

Fig. 1. Example of POST request handler and console output of "request" object

It is important to note that the subset of properties of the "request" object presented here is far from covering even a quarter of all properties inside. Naturally some of those are just methods, others might be circular structures or refer to internal implementation details that are not really relevant to real life scenarios. But things like HTTP headers, Uniform Resource Locator (URL) path, body of the request, parameters, query string parameters, encoding, connection details and others can be really valuable for proper debugging. And as during development it is sometimes not immediately obvious what parts of the incoming client message need deeper investigation, being able to evaluate different computations using dynamic message variant in adaptive logging method can aid considerably.

ISSN 2219-9365

A typical parts of incoming request that can be useful to examine are the original URL and HTTP headers. For a piece of code like “req.headers”, which takes request object in ExpressJS handler and accesses its “headers” property (which is a map from HTTP header name to its value), Figure 2 demonstrates generated abstract syntax tree (note, that some parts are omitted for brevity).



```
{
    "type": "Program",
    "body": [
        {
            "type": "ExpressionStatement",
            "expression": {
                "type": "MemberExpression",
                "object": {
                    "type": "Identifier",
                    "name": "req"
                },
                "property": {
                    "type": "Identifier",
                    "name": "headers"
                }
            }
        }
    ]
}
```


Fig. 2. Abstract syntax tree representation of “req.headers” code expression

This is generated using “acorn” (“GitHub - acornjs/acorn: A small, fast, JavaScript-based JavaScript parser”) Javascript library which is a code parser that outputs hierarchical tree-like structure in Javascript Object Notation (JSON) format. As can be seen from the figure, whole statement is treated as a “program” with one child being an “expression statement”, which in turn consists of two other parts – “object” (meaning an entity which we use to extract a specific property from) and “property” (what “part” of the “object” we want to see). Naturally more sophisticated pieces of code result in more complex ASTs, like function invocations, variable assignments, mathematical operators or even declarations for entire function bodies.

In the context of dynamic messages in the adaptive logging method having this structure is a first step to being able to validate what exactly the invocation will run. It is worth mentioning that parsing code like this only gives some static analysis capabilities and won’t be able to catch reference to variables or properties that do not exist, incorrect use of globals and other errors like that. Those are expected to be caught by runtime implementation, perhaps inside a typical “try … catch” block. With AST structure presented as a JSON object it is already possible to do some decision making and filter out incorrect or unexpected patterns. can be done manually and the resulting tree can then be traversed using Javascript itself, similarly to the task of parsing strings of code - there are better alternatives.

At its core Javascript is a dynamic scripting language with variables that can change type during execution. Because of that a fairly common task is to validate the shape of a particular value and it can be achieved using special validator libraries. One specific library used for validation of Javascript values is called “Ajv” (“Ajv JSON schema validator”) and it functions based on the special “JSON-schema” format (Wright et al., 2022) Those schemas are plain JSON objects themselves but with predefined and predictable structure that allows to describe desired value shape. This definition can then be compiled using Ajv into a special validator and finally the validator function can be called with a parameter we want to check returning false Boolean value if the check failed. Because of the orientation on JSON format, Ajv with JSON-schema format can be seen as a perfect fit for the task of validating JSON-based abstract syntax trees.

The next step in shaping up the necessary parts for validation of dynamic messages of the adaptive logging method is to demonstrate how Ajv and JSON-schema can be used with the aforementioned use case of working with “headers” object of an HTTP request. Figure 2 presents a view of AST generated from “req.headers” code line that is a bit shortened (has no information about positions, for example), but still contains enough structural information to make decisions based on that shape. Mainly it can be observed that a simple “property access” AST has following features: top level element has type “Program” with its body being an array and having exactly 1 element; that

element has type “Expression statement” with its details placed under “expression” key and both parts of that “MemberExpression” object are of type “Identifier”. Figure 3 shows a schema that tests these constraints:

```
const schema = {
  type: 'object', required: ['body', 'type'],
  properties: {
    type: { type: 'string', enum: ['Program'] },
    body: {
      type: 'array', maxItems: 1, minItems: 1,
      items: {
        type: 'object', required: ['expression', 'type'],
        properties: {
          type: { type: 'string', enum: ['ExpressionStatement'] },
          expression: {
            type: 'object', required: ['object', 'property', 'type'],
            properties: {
              type: { type: 'string', enum: ['MemberExpression'] },
              object: {
                type: 'object', required: ['type'],
                properties: { type: { type: 'string', enum: ['Identifier'] } }
              },
              property: {
                type: 'object', required: ['type'],
                properties: { type: { type: 'string', enum: ['Identifier'] } }
              }
}}}}}};
```

Fig. 3. JSON-schema for validation of simple property access AST.

It is worth noting that while being a bit longer than the AST representation in Figure 2, JSON-schema mirrors (to some extent) the nesting of the original, resulting in a declarative and understandable format that has implementations in different programming languages and platforms, can be easily fine-tuned to match required level of precision for a given value, can be combined with logical operations such as “all of” or “any of” and is completely serializable. The last property is essential for a design choice that would allow to finally combine this capability with adaptive logging method.

The introduction of dynamic message capability was accompanied by addition of a new setting in configuration object C (2) – a map that establishes a relationship between dynamic bodies and identifiers used at call sites. An abstract shape of such map is presented in (3):

$$M_{dyn} = Map < string, string > \tag{3}$$

“Map” is used in general programming terms of a mapping from a value of one type to another and in this case the first “string” represents an identifier (not really bound to a specific algorithm or shape, just has to be unique among its neighbors) and second one is the dynamic body itself. With this mapping in place and also considering the fact that validation is directly tied to a specific id in this map (with a possible option that only a subset of all dynamic messages would require corresponding validation logic), a suitable way to store JSON-schemas is in another map (4), which would represent a relation between ids and schema objects ready to be consumed by Ajv and applied when validating a dynamic message that is stored under the same identifier in (3).

$$M_{dyn\,schm} = Map < string, O_{JSON-schema} > \tag{4}$$

where $O_{JSON-schema}$ is an object similar to the one demonstrated in Figure 3.

At this point it becomes possible to perform a final formalization of configuration object C used in (2) and to demonstrate (5) its segments and their relationship:

$$C = \begin{cases} T_{11}^{mod} \cap T_{12}^{mod} \cap \ldots || \, T_{21}^{mod} \cap T_{22}^{mod} \cap \ldots || \ldots \\ M_{dyn} = Map < string, string > \\ M_{dyn\ schm} = Map < string, O_{JSON-schema} > \end{cases} \quad (5)$$

where M_{dyn} and $M_{dyn\ schm}$ were already explained in (3) and (4) respectively and first segment is described in "On specifics of adaptive logging method implementation" and represents a combination of "and" and "or" segments of log tags, forming a set of constraints that can be used to decide whether a particular log invocation should be printed out or skipped (based on its tags provided at call site and result of comparing those to the constraint in configuration). All three of those are grouped using logical operation "and" which conveys the notion of loose relation between segments and further illustrates that the main idea of this research is to expand on basic capabilities of adaptive logging method.

CONCLUSIONS

Ability to have log statements with configurable content (introduced by dynamic message variant) is a very powerful and flexible feature. It adds more tools to properly debug and investigate issues in software development. However, it also presents new challenges related to controlling and restricting what can be executed and how much it can influence other parts of the code base. This paper introduced a potential solution for this issue in a form of validation mechanism based on parsing source code strings into abstract syntax trees and then validating resulting JSON-based structures using Ajv validator and JSON-schema standard. This allows to have a considerable degree of flexibility and precision when adding proper protection around mechanisms of dynamic code execution. Close mapping between AST and schemas simplifies creation of scalable and accurate sets of rules that give a much needed layer of confidence that developed software is more resilient to bugs and unexpected side effects which would otherwise be completely uncontrolled. Possible prospects for further research might look into implementing adaptive logging method, with all accumulated improvements, in real life scenarios such as local development environment or remote cloud-based deployments; it would also be valuable to inspect how far the idea can reach meaning its applicability not only in server based programs written in (mostly) scripting programming languages, but also if it can provide much needed observability improvements in branches such as native development or lambda computations.

References

1. Internet Archive Services Update: 2024-10-17. (2024). Retrieved from: https://blog.archive.org/2024/10/18/internet-archive-services-update-2024-10-17/
2. NVD – CVE-2024-9680. (2024). Retrieved from: https://nvd.nist.gov/vuln/detail/CVE-2024-9680
3. Guri, M. (2024). RAMBO: Leaking Secrets from Air-Gap Computers by Spelling Covert Radio Signals from Computer RAM. *In: Fritsch, L., Hassan, I., Paintsil, E. (eds) Secure IT Systems. NordSec 2023. Lecture Notes in Computer Science, vol 14324. Springer, Cham.* https://doi.org/10.1007/978-3-031-47748-5_9.
4. Microsoft Delays AI-Powered Recall Feature for Copilot+ PCs Amid Security Concerns. (2024). Retrieved from: https://thehackernews.com/2024/06/microsoft-delays-ai-powered-recall.html
5. Ogundipe O., Dr Aweto T. (2024) The shaky foundation of global technology: A case study of the 2024 CrowdStrike outage. *International Journal of Multidisciplinary Research and Growth Evaluation,* vol. 5, issue 5, p. 106-108.
6. Gudimetla S. R. (2024). Cloudstrike Impact on Global Outage and The Challenge of SAAS In the Future. *International Journal of Computer Engineering and Technology*, vol. 15, iss. 4. pp. 472-480. https://doi.org/10.5281/zenodo.13304712.
7. Cleti M., Jano P. (2024). Hallucinations in LLMs: Types, Causes, and Approaches for Enhanced Reliability. https://doi.org/10.13140/RG.2.2.12184.61445
8. Tamburri D. A., Miglierina M., Di Nitto E. (2020). Cloud applications monitoring: An industrial study. *Information and Software Technology, Volume 127, 106376, ISSN 0950-5849.* https://doi.org/10.1016/j.infsof.2020.106376
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. Suprunenko, I., Rudnytskyi, V. (2024). Dynamic message variant in adaptive logging method. *Modern information security, vol. 3, pp. 94-99.* https://doi.org/10.31673/2409-7292.2024.030010
11. Suprunenko, I., Rudnytskyi. (2024). Dynamic source code processing approaches in context of adaptive logging method. *In conference proceedings of the 3rd International Scientific and Practical Internet Conference "Global Society in Formation of New Security System and World Order", July 4-5, pp. 20-22.*

12. Express - Node.js web application framework. Retrieved from: https://expressjs.com/.
13. GitHub - acornjs/acorn: A small, fast, JavaScript-based JavaScript parser. Retrieved from: https://github.com/acornjs/acorn
14. Ajv JSON schema validator. Retrieved from: https://ajv.js.org/
15. Wright A., Andrews H., Hutton B., Dennis G. (2022). JSON Schema: A Media Type for Describing JSON Documents. Retrieved from: https://json-schema.org/draft/2020-12/json-schema-core