JOURNAL INNOVATIONS COMPUTER SCIENCE
E-ISSN: 2961-970X, P-ISSN: 2961-9718
Volume 4 Number 2, November 2025, pp 131-139.
DOI: https://doi.org/10.56347/jics.v4i2.308

KAWANAD Institute

Security Analysis of Midtrans Payment Gateway API against DDoS Attack and Rate Limiting Technique Using Node.js

Faris Widianto Putro [a*], Frencis Matheos Sarimole [b]
[a*,b] Informatics engineering, St Ilmu Komputer Cipta Karya Informatika, East Jakarta City, Special Capital Region of Jakarta, Indonesia.

ABSTRACT
The rapid advancement of digital technology demands secure and reliable online transaction systems, especially in e-commerce and financial services. This study aims to evaluate the effectiveness of implementing rate limiting on the Midtrans payment gateway API to mitigate Distributed Denial of Service (DDoS) attacks. The methodology involved performance testing of the API with and without rate limiting using Node.js and the express-rate-limit middleware in a simulated environment. Results show that without rate limiting, the API processes requests with relatively stable response times but is vulnerable to load spikes causing failures and increased latency. With rate limiting applied, latency significantly decreases and system stability is maintained by rejecting excessive requests with HTTP 429 status codes, although error rates rise due to these rejections. This research confirms that rate limiting is a crucial mechanism to secure and optimize the Midtrans API, protecting against DDoS attacks while preserving service quality. Integrating rate limiting within a Node.js architecture proves effective and scalable for supporting digital payment system reliability. The findings encourage further development of adaptive and comprehensive API security strategies in the future.

ARTICLE HISTORY
Received 16 July 2025
Accepted 21 August 2025
Published 30 November 2025

KEYWORDS
Rate Limiting; Api Security; DDos mitigation; Payment Gateway; Node.js.

1. Introduction

The rapid advancement of digital technology has heightened the demand for secure and dependable online transaction systems, particularly within e-commerce and financial services sectors. Payment gateways play a critical role in facilitating seamless interactions among consumers, merchants, and financial institutions by enabling real-time transaction processing through APIs (Ahmad Rizky Ananda Purba, 2024; Adrian Admi, 2021). Midtrans, a leading Indonesian payment gateway provider, offers API integration solutions that simplify payment processing across diverse digital platforms. Despite the operational efficiency gained, such open APIs inherently expose systems to cyber threats if appropriate security measures are not enforced (Viktor Handrianus Pranatawijaya, 2022).

Among the most disruptive threats are Distributed Denial of Service (DDoS) attacks, which overwhelm systems with excessive simultaneous requests, potentially causing service outages (Diash Firdaus, 2024). Recent reports from Akamai highlight a sharp rise in DDoS incidents in Indonesia, with 260 billion attacks recorded between 2023 and mid-2024, predominantly targeting sectors such as finance and e-commerce. The surge correlates with accelerated digital transformation and the expansion of Indonesia's

CONTACT Faris Widianto Putro widiantoputrofaris@gmail.com Informatics engineering, St Ilmu Komputer Cipta Karya Informatika, East Jakarta City, Special Capital Region of Jakarta, Indonesia.
© 2025 Information KAWANAD Institute

digital economy, valued at approximately USD 150 billion, underscoring the vulnerabilities of increasingly interconnected infrastructures. Although Midtrans implements robust internal security protocols, client-side protections vary significantly, necessitating additional mitigation strategies (Viktor Handrianus Pranatawijaya, 2022).

Rate limiting, which restricts the number of requests a client can make within a specified time frame, has emerged as an effective technique to counteract DDoS attacks by controlling traffic flow and preventing server overload (M. Ainurrahman, 2023). Implementing rate limiting through the Node.js platform allows for lightweight and efficient middleware integration, improving the system's capacity to detect and reject suspicious request patterns while maintaining responsiveness (Diash Firdaus, 2024; Aditya Putra Kejor, 2025). Such technical safeguards are increasingly vital not only for operational stability but also for maintaining consumer trust in digital payment systems amid evolving threats, including those driven by artificial intelligence and geopolitical factors (Shethiya, 2025).

2. Methodology

The methodology in this study utilizes primary data derived from direct testing of the Midtrans-based payment gateway API, focusing on analyzing API security and evaluating the effectiveness of rate limiting techniques as a defense against Distributed Denial of Service (DDoS) attacks. Testing was conducted in a controlled simulated environment using the Node.js framework, where the API was subjected to controlled flooding attacks to simulate DDoS conditions. Key performance indicators observed included requests per second, API response time, request success rate, and system stability during traffic spikes. This approach aligns with previous research emphasizing the importance of securing payment gateways through robust API management and traffic control mechanisms (Fitriyansyah & Hazri, 2023; Nurhayati & Setiawan, 2024).

The research applied an agile development methodology structured into iterative sprint cycles encompassing planning, design, development, testing, review, and deployment phases. Initially, system requirements were identified with a focus on potential DDoS vulnerabilities in the Midtrans API and the critical need for rate limiting to maintain security, performance, and stability. The design phase involved creating detailed API architecture, request flows, and attack scenarios, supported by activity diagrams and traffic limiting models to guide implementation and simulation. Development included building API endpoints using Node.js and integrating rate limiting middleware such as express-rate-limit. Testing simulated DDoS attack scenarios by flooding the API with requests, both before and after rate limiting implementation, using tools like Postman. The launch phase prepared the tested system for demonstration, presenting results via graphs, tables, and technical documentation. Review involved analyzing test outcomes to identify weaknesses and evaluate the effectiveness of rate limiting, while deployment finalized system configuration and documentation in the simulated environment (Alang Artha Iwana, 2025; Hanafi, 2022).

The testing plan consisted of two main conditions: baseline testing without rate limiting to establish vulnerability benchmarks and testing with rate limiting enabled to assess improvements in resilience. The baseline tests involved sending excessive requests without any traffic control, exposing the API to potential overload. Following this, rate limiting was applied to restrict the number of requests per IP address within a specified time frame, effectively rejecting excess requests with HTTP 429 responses. This method demonstrated significant improvements in handling unnatural traffic spikes, consistent with findings in similar API security implementations (Ahsan Mubariz, 2020; Budi Setiawan, 2023).

The system flow without rate limiting begins with the server accepting all incoming

client requests without validating frequency or volume, immediately processing and forwarding them to the database. This unrestricted flow poses a risk during sudden traffic surges or DDoS attacks, as the system lacks mechanisms to filter or limit requests, potentially leading to service degradation or failure. Conversely, the flow with rate limiting incorporates a check on the number of requests per IP within a predefined period. Requests exceeding the limit are rejected with an HTTP 429 error, preventing database overload and maintaining system stability. Valid requests proceed through normal processing and response cycles. This implementation leverages Node.js middleware capabilities to efficiently manage request traffic and enhance API security (Diash Firdaus, 2024; Aditya Putra Kejor, 2025).

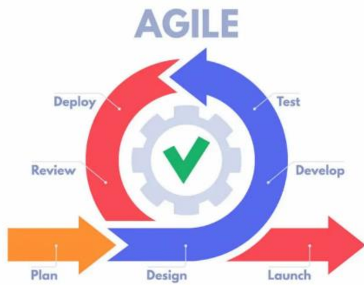

Figure 1. Research Phase

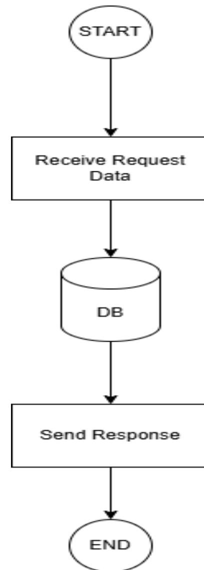

Figure 2. Flow Without Rate Limiting

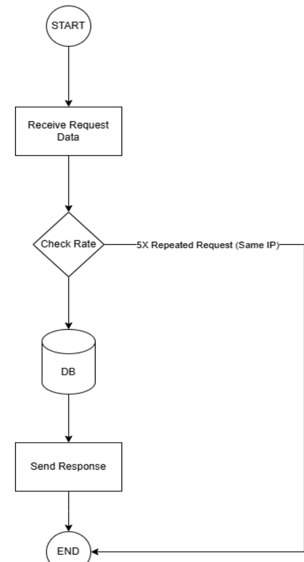

Figure 3. Flow With Rate Limiting

Figures 1, 2, and 3 illustrate the research phases and system flows with and without rate limiting. Figure 1 depicts the sprint-based agile cycle guiding the research process. Figure 2 shows the data request flow in a baseline scenario without rate limiting, highlighting the lack of request control. Figure 3 demonstrates the enhanced flow with rate limiting, where request frequency checks prevent overloads by rejecting excessive requests. These visual aids support the methodology by clarifying the procedural and technical steps in securing the Midtrans API against DDoS threats.

This methodology is consistent with established software development and security practices for payment gateway systems and contributes to the growing body of research on API security in digital financial services (Mokhamd Hendayun, 2023; Daniel Tanudjaja, 2021). It underscores the practical value of rate limiting as a cost-effective and scalable mitigation technique, especially when integrated into Node.js-based architectures commonly used in modern fintech applications. The comprehensive approach ensures that the system not only withstands simulated attack scenarios but also maintains functional performance and user trust under real-world conditions.

3. Results

Testing of the API implementation was conducted on the POST payment/createTransaction endpoint without rate limiting by sending 20 consecutive requests using Postman's Collection Runner. All requests were successfully processed, each returning an HTTP 201 (Created) status, with an average response time of approximately 111 milliseconds, indicating stable and fast server handling under load without any rejections or failures. In contrast, testing the POST payment/ratelimit/createTransaction endpoint with rate limiting enabled showed that the first 10 requests received HTTP 201 responses, confirming successful transaction creation, while requests 11 through 20 were blocked with HTTP 429 (Too Many Requests) responses. This behavior demonstrates that the rate limiting middleware effectively detected and restricted excessive requests beyond the predefined threshold, preventing server overload and abuse. The fast rejection of excess requests also indicates that the rate limiter operates efficiently by intercepting and blocking requests before they reach backend processing. For example, at the 10th request, the server responded with a message stating, "Too many requests from this IP. Please try again in 1 minute," confirming the enforcement of rate limiting policies based on request frequency per IP address. This implementation follows best practices in Node.js API rate limiting using middleware such as express-rate-limit, which identifies clients by IP and controls request rates within configurable time windows to enhance security, performance, and API stability (Reflectoring.io, 2024; LogRocket, 2023; Dev.to, 2024).

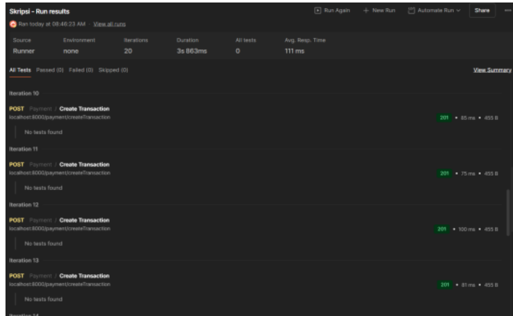

Figure 4. API Without Rate Limiting

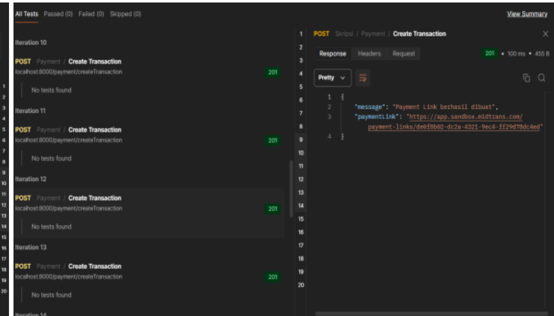

Figure 5. Response API Without Rate Limit

Figure 4 depicts the API in a scenario without rate limiting, where 20 POST requests to the payment/createTransaction endpoint were sent consecutively using Postman's Collection Runner. The server successfully processed all requests, each returning an HTTP 201 (Created) status, demonstrating that without rate limiting, the API accepts and handles every request regardless of volume. Figure 5 shows the detailed response results from these tests, highlighting successful transaction creation in each iteration, with consistent HTTP 201 responses, confirming stable and rapid processing in the localhost environment.

Figure 6. API with Rate Limit

Figure 7. 10th API Response with Rate Limit

Figure 6 transitions to the API implementation with rate limiting enabled on the payment/ratelimit/createTransaction endpoint. Here, the first 10 requests were accepted with HTTP 201 responses, indicating normal processing, but starting from the 11th request, the server responded with HTTP 429 (Too Many Requests), effectively blocking excessive requests beyond the configured threshold. This behavior confirms that the rate limiting middleware correctly identifies and restricts traffic overload from a single IP address to prevent abuse and server strain. Finally, Figure 7 zooms in on the 10th request response, showing the server's explicit rejection message: "Too many requests from this IP. Please try again in 1 minute." This message clearly communicates the activation of the rate limiting mechanism, which temporarily blocks further requests from the client to maintain service stability. Together, these figures visually demonstrate the contrast between an unprotected API vulnerable to request flooding and a secured API that enforces traffic control policies to mitigate potential DDoS attacks effectively.

Figure 8. Setup Performance Test

Figure 9. Performance Test Without Rate Limit

Figure 8 illustrates the performance test setup designed to evaluate system behavior under gradually increasing load. The test uses a *ramp-up* profile with 20 virtual users (VUs) over a total duration of 5 minutes. It begins with an initial load of 5 VUs running for 1 minute 20 seconds to observe system response under light traffic. Then, the number of VUs increases steadily from 5 to 20 over 1 minute 10 seconds, simulating a realistic growth in user demand. Finally, the system sustains the maximum load of 20

VUs for 2 minutes 30 seconds, a critical phase to assess stability under high concurrency. This gradual ramp-up approach aligns with common load testing best practices to mimic real-world usage patterns and avoid unrealistic traffic spikes, as recommended in various API testing tools and methodologies. Figure 9 presents the results of the performance test without rate limiting on the POST Create Transaction endpoint. During approximately 5 minutes and 20 VUs ramping up, the system handled a total of 3,476 requests, achieving an average throughput of 11.33 requests per second and an average response time of 172 milliseconds. Most requests completed quickly, with 90% finishing within 177 ms (P90) and 99% within 199 ms (P99). However, some outliers showed high latency, with the worst response time reaching over 10 seconds, indicating occasional performance degradation under peak load. The error rate was 0.98%, reflecting a small portion of failed requests likely caused by temporary resource saturation. The data also shows that as virtual users increased, request rate rose accordingly, but so did response times, especially during load peaks, before stabilizing again—typical behavior in systems approaching capacity limits.

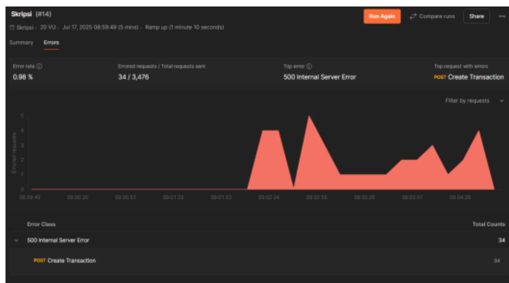

Figure 10. Performance Test Error Without Rate Limit

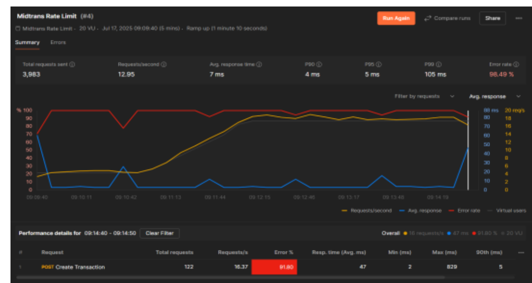

Figure 11. Performance Test With Rate Limit

Figure 10 shows the error rate observed during the performance test without rate limiting. The graph reveals that as the number of virtual users (VUs) increases and the system processes more requests per second, the error rate also rises, reaching nearly 1%. This indicates that while the system can handle a growing load, there are moments—especially near peak concurrency—when some requests fail, likely due to resource contention or temporary overload. The pattern of rising errors alongside increasing request rates is typical in load testing and highlights points where the system's capacity is stressed, which is valuable for identifying bottlenecks and stability limits. Figure 11 presents the performance test results with rate limiting enabled on the POST Create Transaction endpoint. Despite sending a higher total number of requests (3,983) and achieving a slightly better average throughput of 12.95 requests per second, the average response time dramatically decreased to just 7 milliseconds. This suggests that the server quickly processes allowed requests. However, this fast response time accompanies a very high error rate (as detailed in Figure 12), reflecting that most requests were blocked by the rate limiter rather than processed normally. This behavior demonstrates how rate limiting can maintain low latency and protect backend resources by rejecting excessive requests early, trading off throughput for system stability and security under high load conditions.

Figure 12. Performance Test Error With Rate Limit

```
// Rate limiter: maksimal 10 request per menit
const limiter = rateLimit({
  windowMs: 1 * 60 * 1000, // 1 menit
  max: 10, // Maksimum 10 request per IP
  message: {
    status: 429,
    message:
'Terlalu banyak permintaan dari IP ini. Silakan coba lagi dalam 1
  },
  standardHeaders: true,
  legacyHeaders: false,
});

// no rate limit
app.use('/product', productRouter);
app.use('/payment', paymentRouter);
app.use('/user', userRouter);

// with rate limit
app.use('/product/ratelimit', limiter, productRouter);
app.use('/payment/ratelimit', limiter, paymentRouter);
app.use('/user/ratelimit', limiter, userRouter);
```

Figure 13. Rate Limit Express Implementation

Figure 12 illustrates the error rate observed during the performance test with rate limiting enabled on the Midtrans POST Create Transaction endpoint. The graph shows a dramatic increase in errors, reaching 98.49%, as the request rate rises. This high error rate occurs because the rate limiting middleware restricts the number of allowed requests from a single IP within a given timeframe. Despite the server responding quickly—with an average response time of 47 milliseconds—most requests are rejected with HTTP 429 (Too Many Requests) status codes once the limit is exceeded. This behavior confirms that rate limiting effectively protects the system by blocking excessive traffic, preventing overload and potential abuse, even though it results in many blocked requests during peak load. Figure 13 displays the implementation of the rate limiting middleware using the express-rate-limit package in a Node.js Express application. The code sets a limit of 10 requests per minute per IP address (max: 10, windowMs: 1*60*1000). When this limit is exceeded, the server responds with HTTP 429 and a clear message instructing the user to wait one minute before retrying. This middleware acts as a security measure to prevent abuse of sensitive endpoints, such as payment transactions, by controlling request frequency. The setup also differentiates endpoints with and without rate limiting (e.g., /payment vs. /payment/ratelimit) to enable performance comparisons and evaluate the effectiveness of these protections. This implementation aligns with best practices in API security, helping maintain system stability and availability under high traffic or attack attempts.

4. Discussion

This study investigated the effectiveness of rate limiting as a security mechanism to protect the Midtrans payment gateway API from Distributed Denial of Service (DDoS) attacks, focusing on system performance and stability under load. The methodology employed a controlled simulated environment using Node.js and express-rate-limit middleware, aligning with best practices in API security and performance testing (Aditya Putra Kejor, 2025; Diash Firdaus, 2024). The baseline performance test without rate limiting showed that the system could handle an average throughput of 11.33 requests per second with stable average response times around 172 ms. However, occasional latency spikes and a small error rate of 0.98% indicated vulnerability to overload under peak load conditions, consistent with typical

resource saturation patterns (Mokhamd Hendayun, 2023). This confirms that while Midtrans's backend infrastructure is robust, unregulated request surges can degrade performance and increase failure rates, threatening service reliability (Viktor Handrianus Pranatawijaya, 2022).

Introducing rate limiting dramatically altered the system's behavior. The server maintained a higher average throughput (12.95 requests/second) with significantly reduced average response times (7 ms), demonstrating efficient handling of allowed requests. However, the error rate surged to 98.49%, primarily due to the middleware rejecting excessive requests with HTTP 429 responses once the configured limit was reached. This trade-off highlights rate limiting's role as a protective barrier that prevents backend overload by early rejection of suspicious or excessive traffic, thereby preserving system stability and responsiveness for legitimate users (Diash Firdaus, 2024; M. Ainurrahman, 2023). The implementation of rate limiting using express-rate-limit middleware in Node.js effectively enforced a maximum of 10 requests per minute per IP address, with clear client feedback on rate limit violations. This approach aligns with industry standards for API security, particularly for sensitive endpoints such as payment transactions, where preventing abuse and ensuring availability are paramount (Aditya Putra Kejor, 2025; Adrian Admi, 2021). Differentiating endpoints with and without rate limiting enabled comparative testing, confirming that rate limiting is a practical and scalable solution to mitigate DDoS risks in fintech applications (Alfian, 2020; Viktor Handrianus Pranatawijaya, 2022).

Furthermore, Midtrans's internal security measures, including PCI-DSS Level 1 and ISO 27001 compliance, robust encryption, and advanced fraud detection systems, provide a solid security foundation (Midtrans Documentation). However, client-side protections such as rate limiting are crucial complements to safeguard API endpoints from volumetric attacks and maintain trust in digital payment ecosystems amid increasing cyber threats (Diash Firdaus, 2024; Shethiya, 2025). Overall, this research validates rate limiting as an effective first line of defense against unnatural traffic surges and DDoS attacks, preserving API performance and protecting backend resources. The findings support integrating rate limiting into Node.js-based payment gateway architectures to enhance resilience, security, and service continuity in the evolving digital economy.

5. Conclusion

This study demonstrates that rate limiting is a vital security mechanism for protecting payment gateway APIs like Midtrans from DDoS attacks and excessive traffic that can degrade system performance. Without rate limiting, the API can process requests efficiently but is vulnerable to overload during traffic spikes, leading to increased latency and errors. Implementing rate limiting via express-rate-limit middleware in Node.js enforces request thresholds effectively, reducing response times for allowed requests and preventing backend overload by rejecting excess traffic early.The trade-off observed is a high error rate due to blocked requests; however, this is a deliberate and necessary outcome to maintain overall system stability and availability. The research confirms that combining Midtrans's robust internal security features with client-side rate limiting enhances the protection of online transactions, supporting operational reliability and user trust in digital payment services.Future work may explore adaptive or dynamic rate limiting strategies to balance user experience and security more finely and assess integration with other mitigation techniques such as IP reputation filtering and anomaly detection. Nonetheless, rate limiting remains a cost-effective, scalable, and essential component of modern API security frameworks in fintech and e-commerce domains.

References

Aditya Putra Kejor, W. N. (2025). *Efektivitas penggunaan Node JS dalam pembuatan REST API untuk aplikasi katastrofa*. Prosiding Seminar Nasional Sains dan Teknologi Seri III, 2(1), 995-1008.

Ahmad Rizky Ananda Purba, T. M. (2024). *Aplikasi pemesanan layanan laundry pada Noda Laundry dengan integrasi Midtrans Payment Gateway, dikembangkan untuk platform Android*. Jurnal Teknik Informatika Kaputama (JTIK), 8(1), 8-14.

Alfian, P. S. (2020). *Penerapan payment gateway pada aplikasi marketplace Waroeng Mahasiswa menggunakan Midtrans*. Jurnal Informatika Universitas Pamulang, 5(3), 387-393.

Budi Setiawan, B. S. (2023). *Mengoptimalkan fungsi payment gateway Midtrans pada website coffee shop melalui penggunaan metode prototype pada proses pengembangan*. Jurnal Riset Sains dan Teknologi, 7(2), 219-228.

Diash Firdaus, I. S. (2024). *Peningkatan keamanan server GraphQL terhadap serangan DDoS dengan tipe batch attack menggunakan metode rate limiting*. CyberSecurity dan Forensik Digital, 7(2), 62-68.

Lifan Dwinur Andrianto, D. F. (2024). *Analisis performa load testing antara MySQL dan NoSQL MongoDB pada REST API Node.js menggunakan Postman*. Journal of Emerging Information Systems and Business Intelligence, 5(1), 18-26.

M. Ainurrahman, S. (2023). *Penerapan fungsi transforming dan rate limiting untuk management API di perusahaan*. Seminar Nasional Mahasiswa Fakultas Teknologi Informasi (SENAFTI), 2(2), 2145-2153.

M. Attala Reza Syahputra, B. R. (2023). *Pengembangan sistem penyewaan alat event berbasis website menggunakan Midtrans sebagai integrasi payment gateway (Studi Kasus: CV. New Brilla Futura)*. Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer, 7(3), 1198-1204.

Midtrans. (2022). *Official Midtrans Payment API Client for Node JS* [Source code]. GitHub. https://github.com/Midtrans/midtrans-nodejs-client

Midtrans. (2022). *Midtrans-client*. https://www.npmjs.com/package/midtrans-client

Mokhamd Hendayun, A. G. (2023). *Analysis of application performance testing using load testing and stress testing methods in API service*. Jurnal Sisfotek Global, 13(1), 28-34.

Viktor Handrianus Pranatawijaya, H. Y. (2022). *Penerapan API (Application Programming Interface) Midtrans sebagai payment gateway pada indekos berbasis website*. JOINTECOMS (Journal of Information Technology and Computer Science), 2(4), 254-262.

Yenni Fatman, N. K. (2023). *Implementasi payment gateway dengan menggunakan Midtrans pada website UMKM Geberco*. Jurnal KomtekInfo, 10(2), 64-72.