ATTACK SURFACE

How a Visualization and Monitoring Platform Can Expose Your Organization's Data and File System to Attackers

Discovered by HADESS

6 July 2023

WWW.HADESS.IO

Executive Summary

Grafana 7.5.1, a popular open-source analytics and monitoring platform, contains two critical vulnerabilities that can be exploited by attackers. The first vulnerability is a server-side request forgery (SSRF) found in the function `sendWebRequestSync` within the `pkg/services/notification/webhook.go` file. The second vulnerability is a directory traversal issue identified in the function `DownloadFile` within `pkg/cmd/grafana-cli/services/api_client.go`. These vulnerabilities have the potential to expose sensitive information and compromise the security of the application.

The SSRF vulnerability in `sendWebRequestSync` allows attackers to manipulate the URL parameter and make unauthorized requests to internal or external resources. This could lead to data leaks, unauthorized access to sensitive systems, or even compromise of the entire infrastructure.

The directory traversal vulnerability in `DownloadFile` permits attackers to bypass access restrictions and retrieve arbitrary files from the server hosting Grafana. Exploiting this vulnerability enables unauthorized access to sensitive information stored on the server, posing a significant risk to the confidentiality of the data.

To mitigate these vulnerabilities, it is crucial for Grafana users to apply the necessary patches and updates provided by the vendor. Additionally, implementing input validation and sanitization techniques, enforcing strict access controls, and using secure file access methods are recommended best practices to prevent these types of vulnerabilities.

Organizations using Grafana should also conduct regular security assessments, such as code reviews and penetration testing, to identify and address any potential vulnerabilities. Following secure development practices and staying informed about emerging security threats will help ensure the integrity of sensitive data and protect the application from unauthorized access or data breaches.

01

Advisory

A directory traversal vulnerability has been identified in Grafana version 7.5.1, tracked under the CVE-2021-29408 identifier. This vulnerability allows an attacker to manipulate file paths and potentially access sensitive information on the server's file system.

A server-side request forgery (SSRF) vulnerability has been discovered in Grafana version 7.5.1, identified as CVE-2021-29409. This vulnerability enables an attacker to send arbitrary HTTP requests from the affected system, potentially leading to unauthorized access to internal resources or remote code execution.

Abstract

Grafana, a widely used open-source data visualization and monitoring tool, provides a rich set of features for users to create and display metrics, logs, and other data from various sources. However, like any software, Grafana has an attack surface that can be targeted by adversaries. Understanding the attack surface of Grafana is crucial for system administrators and security professionals to effectively protect their deployments.

The attack surface of Grafana can be categorized into several key areas. First, the authentication and access control mechanisms are critical components to secure. Weak or misconfigured authentication can lead to unauthorized access to Grafana instances, potentially exposing sensitive data or allowing attackers to manipulate monitoring systems.

Second, the data sources and integrations supported by Grafana introduce additional attack vectors. This includes databases, APIs, and other external systems that Grafana interacts with to retrieve and display data. Vulnerabilities or misconfigurations in these integrations can lead to data leakage, unauthorized access, or even remote code execution.

Third, plugins and extensions in Grafana extend its functionality but can also introduce security risks. Malicious or vulnerable plugins can be exploited to gain control over the Grafana server or compromise the underlying infrastructure.

Fourth, Grafana's dashboard creation and management capabilities can be abused by attackers. Inadequate input validation or injection vulnerabilities could enable attackers to execute arbitrary code or perform actions with elevated privileges.

Lastly, Grafana's web interface and underlying infrastructure, including the web server and database, form important components of the attack surface. Vulnerabilities in these areas can result in remote code execution, denial-of-service attacks, or unauthorized access to the underlying system.

To mitigate the risks associated with Grafana's attack surface, it is essential to follow security best practices. This includes keeping Grafana and its dependencies up to date, enforcing strong authentication and access control measures, regularly auditing and reviewing plugins and extensions, implementing secure coding practices, and monitoring for suspicious activity or indicators of compromise.

In this context, two notable vulnerabilities in Grafana version 7.5.1 are the Directory Traversal (CVE-2021-29408) and Server-Side Request Forgery (SSRF) (CVE-2021-29409).

1. Directory Traversal (CVE-2021-29408): The Directory Traversal vulnerability in Grafana 7.5.1 (CVE-2021-29408) refers to a security flaw that allows an attacker to access files or directories outside the intended scope. In this specific case, an attacker exploiting this vulnerability could bypass access restrictions and retrieve arbitrary files from the server hosting Grafana. This poses a significant risk as sensitive information stored on the server could be exposed.

2. SSRF (Server-Side Request Forgery) (CVE-2021-29409): The SSRF vulnerability in Grafana 7.5.1 (CVE-2021-29409) is another security issue that can have severe consequences. SSRF occurs when an attacker can make arbitrary HTTP requests from the server hosting the vulnerable application. Exploiting this vulnerability in Grafana could allow an attacker to bypass security controls and access internal resources or services, potentially leading to data breaches, unauthorized access, or further network compromise.

It is important to note that these vulnerabilities were identified in Grafana version 7.5.1 and have since been addressed by the Grafana team. Patches and security fixes were released to mitigate these vulnerabilities. Therefore, it is crucial for users to keep their Grafana installations up to date to ensure protection against known security risks.

02

Technical Analysis

Technical Analysis

Directory Traversal (CVE-2021-29408)

The Directory Traversal vulnerability in Grafana 7.5.1 (CVE-2021-29408) exists in the "DownloadFile" function within the "api_client.go" file located in the "pkg/cmd/grafana-cli/services" directory. This vulnerability allows an attacker to exploit a security flaw, bypass access restrictions, and gain unauthorized access to sensitive information stored on the server hosting Grafana.

The specific cause of the vulnerability lies in the insufficient input validation or sanitization within the "DownloadFile" function. Typically, this function is responsible for handling requests to download files from the Grafana server. However, due to the lack of proper validation, an attacker can manipulate the input parameters to traverse outside the intended directory and access arbitrary files or directories.

The code snippet provided contains several functions that interact with a Grafana server for retrieving and managing plugins. The vulnerability arises in the `DownloadFile` function, where a directory traversal vulnerability can be exploited if proper input validation and sanitization are not implemented.

```
func (client *GrafanaComClient) GetPlugin(pluginId, repoUrl string) (models.Plugin, error) {
 logger.Debugf("getting plugin metadata from: %v pluginId: %v \n", repoUrl, pluginId)
 body, err := sendRequestGetBytes(HttpClient, repoUrl, "repo", pluginId)
 if err != nil {
  if errors.Is(err, ErrNotFoundError) {
     return models.Plugin{}, errutil.Wrap("Failed to find requested plugin, check if the
plugin_id is correct", err)
  }
  return models.Plugin{}, errutil.Wrap("Failed to send request", err)
 }

 var data models.Plugin
 err = json.Unmarshal(body, &data)
 if err != nil {
  logger.Info("Failed to unmarshal plugin repo response error:", err)
  return models.Plugin{}, err
 }

 return data, nil
}
```

```
func (client *GrafanaComClient) DownloadFile(pluginName string, tmpFile
*os.File, url string, checksum string) (err error) {
 // Try handling URL as a local file path first
 if _, err := os.Stat(url); err == nil {
  // We can ignore this gosec G304 warning since `url` stems from command
line flag "pluginUrl". If the
  // user shouldn't be able to read the file, it should be handled through
filesystem permissions.
  // nolint:gosec
  f, err := os.Open(url)
  if err != nil {
   return errutil.Wrap("Failed to read plugin archive", err)
  }
  _, err = io.Copy(tmpFile, f)
  if err != nil {
   return errutil.Wrap("Failed to copy plugin archive", err)
  }
  return nil
 }

 client.retryCount = 0

 defer func() {
  if r := recover(); r != nil {
   client.retryCount++
   if client.retryCount < 3 {
    logger.Info("Failed downloading. Will retry once.")
    err = tmpFile.Truncate(0)
    if err != nil {
     return
    }
    _, err = tmpFile.Seek(0, 0)
    if err != nil {
     return
    }
    err = client.DownloadFile(pluginName, tmpFile, url, checksum)
   } else {
    client.retryCount = 0
    failure := fmt.Sprintf("%v", r)
    if failure == "runtime error: makeslice: len out of range" {
        err = fmt.Errorf("corrupt HTTP response from source, please try
again")
    } else {
     panic(r)
    }
   }
  }
 }()
```

The `DownloadFile` function accepts a plugin name, a temporary file to store the downloaded file, a URL pointing to the file, and a checksum for integrity validation. The function begins by attempting to handle the URL as a local file path. If the file exists locally, it is opened and copied to the temporary file, which is later used for further processing.

```
if _, err := os.Stat(url); err == nil {
    f, err := os.Open(url)
    if err != nil {
        return errutil.Wrap("Failed to read plugin archive", err)
    }
    _, err = io.Copy(tmpFile, f)
    if err != nil {
        return errutil.Wrap("Failed to copy plugin archive", err)
    }
    return nil
}
```

The code first attempts to handle the URL as a local file path by checking if the file exists using `os.Stat`. If the file is found, it is opened and its contents are copied to the temporary file. However, this approach is vulnerable to a directory traversal attack. An attacker can manipulate the `url` parameter by providing a path that includes traversal sequences (`../` or `../../`), allowing them to bypass access restrictions and read arbitrary files on the server.

To address this vulnerability and achieve compliance with secure coding practices, the following modifications can be made to the code:

1. Input Validation: Implement robust input validation to ensure that the `url` parameter refers only to the intended file within the Grafana server's file system. This can be done by validating the URL format and ensuring it points to a specific location within the expected directory structure. Reject any URLs that contain traversal sequences or attempt to access files outside the designated directory.

2. Use Safe File Access Methods: Instead of directly opening and copying files based on the URL provided, utilize safer file access methods. For example, instead of using `os.Open` on the URL, prefer using `os.OpenFile` with the appropriate flags and permissions. This allows for better control and validation of the file access.

3. Apply Whitelisting or Restricted Access: Maintain a whitelist of allowed file paths or a restricted access mechanism to limit the files that can be accessed. This ensures that only authorized files are accessible and prevents potential exposure of sensitive information.

The impact of this vulnerability can be severe, as it exposes sensitive information that can be leveraged for further attacks or exploitation. An attacker who successfully exploits this vulnerability can gain unauthorized access to confidential data, compromise user accounts, or obtain sensitive system information.

To mitigate the risk associated with this vulnerability, it is crucial to apply the security patch or upgrade to a version of Grafana that includes the fix provided by the Grafana team. This fix typically involves implementing proper input validation and sanitization techniques within the "DownloadFile" function to prevent directory traversal attacks.

SSRF in Grafana 7.5.1(CVE-2021-29409)

Server-Side Request Forgery (SSRF) is a critical security vulnerability that allows an attacker to manipulate an application's server-side requests to make unauthorized requests to internal or external resources. This article will provide a deep analysis of SSRF vulnerabilities in Golang applications and discuss the potential impact of such vulnerabilities. We will also explore common mitigation techniques and best practices to secure Golang applications against SSRF attacks.

An SSRF vulnerability arises when an application does not properly validate or restrict user-controlled input used in making server-side requests. Attackers exploit this vulnerability by manipulating the input to redirect requests to unintended targets, such as internal network resources, sensitive information repositories, or external systems.

In this code snippet, the `sendWebRequestSync` function constructs an HTTP request using user-controlled input (`webhook.Url`). Without proper validation and restriction, an attacker can manipulate the URL to target unintended resources.

The attacker leverages the well-known metadata service IP address (169.254.169.254), commonly available within cloud environments. By making a request to this URL, the attacker attempts to retrieve sensitive security credentials of the underlying infrastructure. If the application executing this code is running within the cloud environment and has access to this metadata service, the SSRF vulnerability allows the attacker to exfiltrate these credentials.

```
func (ns *NotificationService) sendWebRequestSync(ctx context.Context, webhook *Webhook) error {
 ns.log.Debug("Sending webhook", "url", webhook.Url, "http method", webhook.HttpMethod)

 if webhook.HttpMethod == "" {
  webhook.HttpMethod = http.MethodPost
 }

 if webhook.HttpMethod != http.MethodPost && webhook.HttpMethod != http.MethodPut {
  return fmt.Errorf("webhook only supports HTTP methods PUT or POST")
 }

 request, err := http.NewRequest(webhook.HttpMethod, webhook.Url, bytes.NewReader([]byte(webhook.Body)))
 if err != nil {
  return err
 }

 if webhook.ContentType == "" {
  webhook.ContentType = "application/json"
 }

 request.Header.Set("Content-Type", webhook.ContentType)
 request.Header.Set("User-Agent", "Grafana")

 if webhook.User != "" && webhook.Password != "" {
  request.Header.Set("Authorization", util.GetBasicAuthHeader(webhook.User, webhook.Password))
 }

 for k, v := range webhook.HttpHeader {
  request.Header.Set(k, v)
 }

 resp, err := ctxhttp.Do(ctx, netClient, request)
 if err != nil {
  return err
 }
 defer func() {
  if err := resp.Body.Close(); err != nil {
   ns.log.Warn("Failed to close response body", "err", err)
  }
 }()

 if resp.StatusCode/100 == 2 {
  ns.log.Debug("Webhook succeeded", "url", webhook.Url, "statuscode", resp.Status)
  // flushing the body enables the transport to reuse the same connection
  if _, err := io.Copy(ioutil.Discard, resp.Body); err != nil {
   ns.log.Error("Failed to copy resp.Body to ioutil.Discard", "err", err)
  }
  return nil
 }

 body, err := ioutil.ReadAll(resp.Body)
 if err != nil {
  return err
 }

 ns.log.Debug("Webhook failed", "url", webhook.Url, "statuscode", resp.Status, "body", string(body))
 return fmt.Errorf("Webhook response status %v", resp.Status)
}
```

The code snippet provided contains the `sendWebRequestSync` function, which is responsible for sending a synchronous web request to a specified URL. However, this implementation is susceptible to a Server-Side Request Forgery (SSRF) vulnerability, where an attacker can manipulate the URL to make unauthorized requests to internal or external resources.

```
request, err := http.NewRequest(webhook.HttpMethod, webhook.Url, bytes.NewReader([]byte(webhook.Body)))
if err != nil {
   return err
}

// ...

resp, err := ctxhttp.Do(ctx, netClient, request)
if err != nil {
   return err
}
```

The code constructs an HTTP request using the `http.NewRequest` function, with the URL provided in the `webhook.Url` variable. This allows an attacker to control the URL and potentially send requests to unintended destinations. The code does not implement any validation or restrictions on the URL, which leaves the application vulnerable to SSRF attacks.

To address this vulnerability and achieve compliance with secure coding practices, the following modifications can be made to the code:

1. URL Whitelisting: Maintain a whitelist of trusted URLs that the application is allowed to access. Validate the `webhook.Url` against this whitelist before making the HTTP request. Reject any URLs that are not in the whitelist or do not meet the expected pattern.

2. URL Validation and Sanitization: Implement strict validation and sanitization techniques on the `webhook.Url` to ensure it points only to the intended resources. Use a well-tested URL parsing library or regular expressions to validate the URL's scheme, host, and path components. Additionally, consider removing any unnecessary or potentially dangerous URL components, such as fragment identifiers or query parameters.

3. Network Access Controls: Consider implementing network access controls within the application's infrastructure. This can involve using firewalls, security groups, or network segmentation to restrict outbound requests from the application server to trusted resources only. Block or limit access to sensitive internal systems or external services that are not essential for the application's functionality.

4. Implement Whitelisted HTTP Methods: Restrict the allowed HTTP methods for the web requests. Define a whitelist of acceptable HTTP methods that the application can use, such as POST or PUT, and reject any other methods.

03

Conclusion

The presence of server-side request forgery (SSRF) and directory traversal vulnerabilities in Grafana 7.5.1 raises significant concerns regarding the security of the application. These vulnerabilities can lead to unauthorized access to sensitive information and pose a risk to the confidentiality, integrity, and availability of the system.

In the function `sendWebRequestSync` within the `pkg/services/notification/webhook.go` file, the SSRF vulnerability allows attackers to manipulate the URL and potentially make unauthorized requests to internal or external resources. This can result in data leakage, unauthorized access to sensitive systems or APIs, and potential compromise of the application's infrastructure.

Additionally, the directory traversal vulnerability in the `DownloadFile` function within `pkg/cmd/grafana-cli/services/api_client.go` exposes the application to the risk of unauthorized file access. Attackers can exploit this vulnerability to bypass access restrictions and retrieve arbitrary files from the server hosting Grafana. This can lead to the exposure of sensitive information stored on the server.

To mitigate these vulnerabilities, it is crucial to implement secure coding practices and follow recommended guidelines. This includes implementing input validation and sanitization techniques to prevent SSRF attacks, enforcing strict access controls, and using safe file access methods to prevent directory traversal exploits. Regularly updating the Grafana software and its dependencies with the latest security patches is also essential to address known vulnerabilities.

cat ~/.hadess

We are "Hadess"; A group of cyber security experts and white hat hackers who, in addition to discovering and reporting vulnerabilities to big companies such as Google, Apple and Twitter, have the honor of working with famous Iranian companies over the past years. HADESS Company provides its customers with integrated solutions in the field of cyber security, with a deep insight and understanding of the software development process as well as the development infrastructure.

Website:

WWW.HADESS.IO

Email

MARKETING@HADESS.IO