The New OSINT Battlefield Is the Open-Source Developer

Maria Cattini | 08/06/2026 | CYBERSECURITY

A compromised developer account does not look like an intrusion at first.

It may look like a normal commit. A routine extension update. A new package version. A dependency added to a manifest. A GitHub push from an account that already belongs to the project.

That is the problem.

In May 2026, CrowdStrike said it coordinated with Google and the Shadowserver Foundation to disrupt Glassworm, a botnet that targeted software developers through the open-source supply chain. The operation struck four command-and-control channels at the same time and cut the operators off from infected machines.

For OSINT analysts, the important lesson is not only that a botnet was taken down.

The lesson is that modern supply-chain investigations increasingly start with developers, repositories, package registries, extensions, tokens, commit history and infrastructure that was designed to look trusted.

Open source is not only code.

It is an ecosystem of accounts, permissions, habits and implicit trust.

What Glassworm Shows

According to CrowdStrike, Glassworm operators had targeted software developers since at least early 2025. The campaign used several delivery routes: trojanized VS Code extensions published on Open VSX, compromised npm and Python packages, and poisoned GitHub repositories pushed through stolen developer credentials.

CrowdStrike said more than 300 GitHub repositories were poisoned during the campaign. The same report described credential theft, information stealing and a Node.js remote access tool called GlasswormRAT. The operation affected Windows, macOS and Linux systems.

TechCrunch reported the same takedown and highlighted the central trust problem: attackers were not only targeting software products. They were targeting the people and accounts that build them.

That distinction matters.

When a malicious package is published by a new unknown account, the risk is visible. When malicious code is pushed through a developer account that already has access, the signal is weaker. The repository may have history. The package may have users. The extension may have downloads. The maintainer may be real.

The attacker is borrowing trust.

Why Developers Are High-Value Targets

Developers sit close to the systems that organizations depend on.

They may have access to:

• source code repositories;
• package registries;
• CI/CD pipelines;
• cloud environments;
• signing keys;
• API tokens;
• npm, PyPI or Open VSX publishing credentials;
• internal documentation;
• build and deployment workflows.

Compromising one developer workstation or one publisher account can create a path into many downstream environments.

This is why a developer-targeting campaign has a different blast radius from a normal endpoint infection. The first victim may be one person. The second victim may be every organization that installs, builds, imports or trusts the code that person can publish.

From an OSINT point of view, this changes the investigation.

You are not only looking for a malicious file. You are looking for a trust path.

The Four Surfaces to Map First

When investigating an open-source supply-chain incident, start by mapping four surfaces.

1. The Repository Surface

The repository is the obvious place to start, but it should not be treated as a flat code folder.

Look at:

• recent commits;
• force-push activity;
• new contributors or unusual author metadata;
• changes to default branches;
• edits to installation scripts, build files and release workflows;
• binary or non-printable characters in changed files;
• sudden changes in dependency manifests;
• commit messages that feel generic, vague or disconnected from the code change.

Do not assume that a familiar account means a safe commit.

In account hijacking cases, the account is part of the attack surface.

2. The Package Surface

Package registries are where trust becomes distribution.

Check:

• new versions published close to the incident window;
• changes to postinstall, setup.py, build scripts or package lifecycle hooks;
• maintainers added or removed;
• package names that imitate trusted tools;
• dependencies added in a minor update;
• sudden cross-publishing across ecosystems;
• mismatch between repository code and published package content.

The last point is often overlooked. A GitHub repository may look clean while the package distributed through a registry contains different material. Always compare the source repository, release artifact and package registry metadata when possible.

3. The Extension Surface

Developer extensions are high-trust tools.

They run inside environments where code, credentials and project context are already present. In the Glassworm case, the campaign abused VS Code and Open VSX extension ecosystems, including trojanized extensions and transitive delivery techniques.

For extension analysis, check:

• publisher history;
• recent ownership or account changes;
• new dependencies declared in extension manifests;
• permissions requested by the extension;
• runtime network behavior;
• updates published after long inactivity;
• extensions that impersonate popular developer tools;
• differences between marketplace description and actual package behavior.

Download count and age are not enough.

If an established publisher account is compromised, those trust signals become part of the lure.

4. The Infrastructure Surface

Glassworm is also useful because its command-and-control infrastructure was built for resilience.

CrowdStrike described four C2 channels: Solana blockchain memo fields, BitTorrent DHT, Google Calendar event titles and traditional VPS infrastructure. The point is not that every analyst must reverse-engineer malware infrastructure. The point is that attackers can hide coordination inside legitimate or decentralized services that are hard to block with simple domain rules.

For an OSINT workflow, document:

• domains and IPs;
• wallet addresses or blockchain artifacts, if relevant;
• calendar or public-service abuse patterns;
• GitHub-hosted payload links;
• infrastructure reuse across versions;
• timestamps of infrastructure changes;
• relationships between infrastructure and package publication events.

The goal is not to chase every technical detail.

The goal is to understand how the operation stayed alive.

A Practical OSINT Workflow

A defensive open-source supply-chain investigation should move in layers.

Step 1: Define the Incident Window

Do not start with every repository, every package and every extension at once.

Start with time.

Ask:

• when was the suspicious package version published?
• when was the extension updated?
• when did the unusual commit appear?
• when did credentials appear to be used?
• when did external reports first mention the campaign?
• when did the takedown or remediation occur?

A timeline prevents the investigation from becoming a list of disconnected artifacts.

Step 2: Separate Account Activity From Code Activity

A malicious commit and a compromised account are related, but they are not the same evidence.

Track them separately:

• account login or token activity;
• commit metadata;
• package publishing events;
• pull requests and issue activity;
• changes to collaborators;
• registry maintainers;
• release tags;
• CI/CD workflow runs.

This helps avoid a common mistake: assuming that because a trusted account performed an action, the action itself is trusted.

Step 3: Compare Expected Behavior With Observed Behavior

Every project has habits.

Some maintainers publish weekly. Others publish rarely. Some use signed commits. Others do not. Some projects have formal releases. Others push directly.

Compare the suspicious activity with the project’s normal pattern:

• Is the commit style consistent?
• Is the release frequency normal?
• Is the package versioning pattern coherent?
• Is the maintainer using the same account and signing behavior?
• Did the change happen at an unusual time?

• Did the update touch sensitive files without explanation?

OSINT is often strongest when it uses public metadata to detect a break in routine.

Step 4: Inspect the Dependency Chain

Do not stop at the first package.

Map:

• direct dependencies;
• transitive dependencies;
• extension dependencies;
• package lifecycle scripts;
• external URLs fetched at install or runtime;
• GitHub Actions or CI/CD dependencies;
• container images, if used;
• release artifacts not visible in source code.

Glassworm-style campaigns show why this matters. The malicious element may not sit in the package the developer intentionally installed. It may arrive through an extension dependency, a package lifecycle hook or a compromised account that publishes an update to something already trusted.

Step 5: Preserve Evidence Before It Disappears

Open-source incidents move fast.

Packages are removed. Repositories are cleaned. Extensions are unpublished. Accounts are suspended. Domains are sinkholed. Marketplace pages vanish.

Preserve:

• package metadata;
• release version lists;
• repository commits;
• screenshots of marketplace pages;
• maintainer lists;
• dependency manifests;
• URLs and timestamps;
• archived copies where legally and ethically appropriate;
• hashes of downloaded artifacts, if you are authorized to handle them safely.

Do not download or execute suspicious code on a normal workstation.

If you need malware analysis, use an isolated environment and follow your organization’s procedures. If you do not have that environment, stop at metadata and public reporting.

Checklist: What to Review After a Developer-Targeting Campaign

Use this as a defensive triage checklist.

Repository Checks

• Review recent commits to default branches.

• Look for force-push events.
• Check new or unexpected contributors.
• Search for changes to install, build and release scripts.
• Inspect dependency manifest changes.
• Look for non-printable or unusual Unicode characters in code diffs.
• Compare GitHub source with published packages.

Account Checks

• Review GitHub audit logs, if available.
• Check personal access token creation and use.
• Rotate tokens if compromise is suspected.
• Verify MFA status for maintainers.
• Review added collaborators, deploy keys and webhooks.
• Check whether the same credentials had registry publishing access.

Package Checks

• Review npm, PyPI or other registry publication history.
• Compare package contents against source repositories.
• Inspect lifecycle hooks and setup scripts.
• Look for newly added dependencies.
• Check maintainers and ownership changes.
• Confirm whether the package was published from expected infrastructure.

Extension Checks

• Audit installed VS Code and compatible editor extensions.
• Review publisher history and recent updates.
• Check extension dependencies and extension packs.
• Disable automatic extension updates where policy requires review.
• Investigate extensions that fetch remote code at runtime.
• Maintain an allowlist for developer environments.

Infrastructure Checks

• Review unexpected outbound connections from developer workstations.
• Investigate unusual access to blockchain RPC endpoints where not expected.
• Look for non-browser access patterns to public services such as calendar APIs.
• Map relationships between infrastructure, package updates and repository commits.
• Treat one indicator as a lead, not a conclusion.

Common Mistakes

The first mistake is trusting popularity.

A package with many users can still be compromised. An old extension can still receive a malicious update. A real maintainer can still lose an account.

The second mistake is looking only at code.

Supply-chain attacks often live in metadata: who published, when, from where, under which account, with which token, into which registry, after which previous event.

The third mistake is treating removal as resolution.

If a malicious package is removed, the investigation is not over. Tokens may still be compromised. Forks may still exist. Cached artifacts may remain. Downstream builds may have already consumed the version.

The fourth mistake is over-attribution.

Locale checks, language comments, infrastructure choices and code style can all support an assessment, but none should be treated as proof alone. CrowdStrike itself noted that no single indicator is proof on its own.

The fifth mistake is turning the article into an attack manual.

For public OSINT work, the useful output is not a recipe for reproducing malware behavior. It is a defensible map of exposure, trust paths, affected surfaces and remediation priorities.

Why This Matters for Non-Developers Too

Many organizations do not think of themselves as software companies.

They still consume software.

They use open-source libraries, browser extensions, developer tools, cloud scripts, automation workflows, containers, dashboards and internal tools built from public components.

That means they inherit trust from people they have never met.

An open-source developer’s account may sit several layers away from the final organization, but the dependency chain can bring that risk inside the environment. This is why supply-chain OSINT is becoming more important: the question is no longer only "is this file malicious?".

The better question is:

which trust relationship allowed this code to reach users?

That question leads to accounts, repositories, registries, extensions, dependencies and infrastructure.

It also leads to better defense.

The Takeaway

Glassworm is not only a story about a botnet.

It is a reminder that the open-source supply chain is an investigation surface.

The developer is part of that surface. So is the extension marketplace. So is the package registry. So is the CI/CD pipeline. So is the command-and-control layer hidden behind legitimate or decentralized services.

For OSINT analysts, the task is to connect these layers without overstating certainty.

Start with the timeline. Separate account activity from code activity. Map the dependency chain. Preserve evidence. Treat public metadata as evidence, not decoration. And never assume that a trusted account means a trusted action.

In modern supply-chain incidents, the most important clue may not be the malware.

It may be the path that made the malware look ordinary.
A compromised developer account does not look like an intrusion at first.

It may look like a normal commit. A routine extension update. A new package version. A dependency added to a manifest. A GitHub push from an account that already belongs to the project.

That is the problem.

In May 2026, CrowdStrike said it coordinated with Google and the Shadowserver Foundation to disrupt Glassworm, a botnet that targeted software developers through the open-source supply chain. The operation struck four command-and-control channels at the same time and cut the operators off from infected machines.

For OSINT analysts, the important lesson is not only that a botnet was taken down.

The lesson is that modern supply-chain investigations increasingly start with developers, repositories, package registries, extensions, tokens, commit history and infrastructure that was designed to look trusted.

Open source is not only code.

It is an ecosystem of accounts, permissions, habits and implicit trust.

What Glassworm Shows

According to CrowdStrike, Glassworm operators had targeted software developers since at least early 2025. The campaign used several delivery routes: trojanized VS Code extensions published on Open VSX, compromised npm and Python packages, and poisoned GitHub repositories pushed through stolen developer credentials.

CrowdStrike said more than 300 GitHub repositories were poisoned during the campaign. The same report described credential theft, information stealing and a Node.js remote access tool called GlasswormRAT. The operation affected Windows, macOS and Linux systems.

TechCrunch reported the same takedown and highlighted the central trust problem: attackers were not only targeting software products. They were targeting the people and accounts that build them.

That distinction matters.

When a malicious package is published by a new unknown account, the risk is visible. When malicious code is pushed through a developer account that already has access, the signal is weaker. The repository may have history. The package may have users. The extension may have downloads. The maintainer may be real.

The attacker is borrowing trust.

Why Developers Are High-Value Targets

Developers sit close to the systems that organizations depend on.

They may have access to:

- source code repositories;
- package registries;
- CI/CD pipelines;
- cloud environments;
- signing keys;
- API tokens;
- npm, PyPI or Open VSX publishing credentials;
- internal documentation;
- build and deployment workflows.

Compromising one developer workstation or one publisher account can create a path into many downstream environments.

This is why a developer-targeting campaign has a different blast radius from a normal endpoint infection. The first victim may be one person. The second victim may be every organization that installs, builds, imports or trusts the code that person can publish.

From an OSINT point of view, this changes the investigation.

You are not only looking for a malicious file. You are looking for a trust path.

The Four Surfaces to Map First

When investigating an open-source supply-chain incident, start by mapping four surfaces.

1. The Repository Surface

The repository is the obvious place to start, but it should not be treated as a flat code folder.

Look at:

• recent commits;
• force-push activity;
• new contributors or unusual author metadata;
• changes to default branches;
• edits to installation scripts, build files and release workflows;
• binary or non-printable characters in changed files;
• sudden changes in dependency manifests;
• commit messages that feel generic, vague or disconnected from the code change.

Do not assume that a familiar account means a safe commit.

In account hijacking cases, the account is part of the attack surface.

2. The Package Surface

Package registries are where trust becomes distribution.

Check:

• new versions published close to the incident window;
• changes to postinstall, setup.py, build scripts or package lifecycle hooks;
• maintainers added or removed;
• package names that imitate trusted tools;
• dependencies added in a minor update;
• sudden cross-publishing across ecosystems;
• mismatch between repository code and published package content.

The last point is often overlooked. A GitHub repository may look clean while the package distributed through a registry contains different material. Always compare the source repository, release artifact and package registry metadata when possible.

3. The Extension Surface

Developer extensions are high-trust tools.

They run inside environments where code, credentials and project context are already present. In

the Glassworm case, the campaign abused VS Code and Open VSX extension ecosystems, including trojanized extensions and transitive delivery techniques.

For extension analysis, check:

• publisher history;
• recent ownership or account changes;
• new dependencies declared in extension manifests;
• permissions requested by the extension;
• runtime network behavior;
• updates published after long inactivity;
• extensions that impersonate popular developer tools;
• differences between marketplace description and actual package behavior.

Download count and age are not enough.

If an established publisher account is compromised, those trust signals become part of the lure.

4. The Infrastructure Surface

Glassworm is also useful because its command-and-control infrastructure was built for resilience.

CrowdStrike described four C2 channels: Solana blockchain memo fields, BitTorrent DHT, Google Calendar event titles and traditional VPS infrastructure. The point is not that every analyst must reverse-engineer malware infrastructure. The point is that attackers can hide coordination inside legitimate or decentralized services that are hard to block with simple domain rules.

For an OSINT workflow, document:

• domains and IPs;
• wallet addresses or blockchain artifacts, if relevant;
• calendar or public-service abuse patterns;
• GitHub-hosted payload links;
• infrastructure reuse across versions;
• timestamps of infrastructure changes;
• relationships between infrastructure and package publication events.

The goal is not to chase every technical detail.

The goal is to understand how the operation stayed alive.

A Practical OSINT Workflow

A defensive open-source supply-chain investigation should move in layers.

Step 1: Define the Incident Window

Do not start with every repository, every package and every extension at once.

Start with time.

Ask:

• when was the suspicious package version published?
• when was the extension updated?
• when did the unusual commit appear?
• when did credentials appear to be used?

• when did external reports first mention the campaign?
• when did the takedown or remediation occur?

A timeline prevents the investigation from becoming a list of disconnected artifacts.

Step 2: Separate Account Activity From Code Activity

A malicious commit and a compromised account are related, but they are not the same evidence.

Track them separately:

• account login or token activity;
• commit metadata;
• package publishing events;
• pull requests and issue activity;
• changes to collaborators;
• registry maintainers;
• release tags;
• CI/CD workflow runs.

This helps avoid a common mistake: assuming that because a trusted account performed an action, the action itself is trusted.

Step 3: Compare Expected Behavior With Observed Behavior

Every project has habits.

Some maintainers publish weekly. Others publish rarely. Some use signed commits. Others do not. Some projects have formal releases. Others push directly.

Compare the suspicious activity with the project’s normal pattern:

• Is the commit style consistent?
• Is the release frequency normal?
• Is the package versioning pattern coherent?
• Is the maintainer using the same account and signing behavior?
• Did the change happen at an unusual time?
• Did the update touch sensitive files without explanation?

OSINT is often strongest when it uses public metadata to detect a break in routine.

Step 4: Inspect the Dependency Chain

Do not stop at the first package.

Map:

• direct dependencies;
• transitive dependencies;
• extension dependencies;
• package lifecycle scripts;
• external URLs fetched at install or runtime;
• GitHub Actions or CI/CD dependencies;
• container images, if used;
• release artifacts not visible in source code.

Glassworm-style campaigns show why this matters. The malicious element may not sit in the package the developer intentionally installed. It may arrive through an extension dependency, a package lifecycle hook or a compromised account that publishes an update to something already trusted.

Step 5: Preserve Evidence Before It Disappears

Open-source incidents move fast.

Packages are removed. Repositories are cleaned. Extensions are unpublished. Accounts are suspended. Domains are sinkholed. Marketplace pages vanish.

Preserve:

- package metadata;
- release version lists;
- repository commits;
- screenshots of marketplace pages;
- maintainer lists;
- dependency manifests;
- URLs and timestamps;
- archived copies where legally and ethically appropriate;
- hashes of downloaded artifacts, if you are authorized to handle them safely.

Do not download or execute suspicious code on a normal workstation.

If you need malware analysis, use an isolated environment and follow your organization’s procedures. If you do not have that environment, stop at metadata and public reporting.

Checklist: What to Review After a Developer-Targeting Campaign

Use this as a defensive triage checklist.

Repository Checks

- Review recent commits to default branches.
- Look for force-push events.
- Check new or unexpected contributors.
- Search for changes to install, build and release scripts.
- Inspect dependency manifest changes.
- Look for non-printable or unusual Unicode characters in code diffs.
- Compare GitHub source with published packages.

Account Checks

- Review GitHub audit logs, if available.
- Check personal access token creation and use.
- Rotate tokens if compromise is suspected.
- Verify MFA status for maintainers.
- Review added collaborators, deploy keys and webhooks.
- Check whether the same credentials had registry publishing access.

Package Checks

- Review npm, PyPI or other registry publication history.
- Compare package contents against source repositories.

• Inspect lifecycle hooks and setup scripts.
• Look for newly added dependencies.
• Check maintainers and ownership changes.
• Confirm whether the package was published from expected infrastructure.

Extension Checks

• Audit installed VS Code and compatible editor extensions.
• Review publisher history and recent updates.
• Check extension dependencies and extension packs.
• Disable automatic extension updates where policy requires review.
• Investigate extensions that fetch remote code at runtime.
• Maintain an allowlist for developer environments.

Infrastructure Checks

• Review unexpected outbound connections from developer workstations.
• Investigate unusual access to blockchain RPC endpoints where not expected.
• Look for non-browser access patterns to public services such as calendar APIs.
• Map relationships between infrastructure, package updates and repository commits.
• Treat one indicator as a lead, not a conclusion.

Common Mistakes

The first mistake is trusting popularity.

A package with many users can still be compromised. An old extension can still receive a malicious update. A real maintainer can still lose an account.

The second mistake is looking only at code.

Supply-chain attacks often live in metadata: who published, when, from where, under which account, with which token, into which registry, after which previous event.

The third mistake is treating removal as resolution.

If a malicious package is removed, the investigation is not over. Tokens may still be compromised. Forks may still exist. Cached artifacts may remain. Downstream builds may have already consumed the version.

The fourth mistake is over-attribution.

Locale checks, language comments, infrastructure choices and code style can all support an assessment, but none should be treated as proof alone. CrowdStrike itself noted that no single indicator is proof on its own.

The fifth mistake is turning the article into an attack manual.

For public OSINT work, the useful output is not a recipe for reproducing malware behavior. It is a defensible map of exposure, trust paths, affected surfaces and remediation priorities.

Why This Matters for Non-Developers Too

Many organizations do not think of themselves as software companies.

They still consume software.

They use open-source libraries, browser extensions, developer tools, cloud scripts, automation workflows, containers, dashboards and internal tools built from public components.

That means they inherit trust from people they have never met.

An open-source developer’s account may sit several layers away from the final organization, but the dependency chain can bring that risk inside the environment. This is why supply-chain OSINT is becoming more important: the question is no longer only "is this file malicious?".

The better question is:

which trust relationship allowed this code to reach users?

That question leads to accounts, repositories, registries, extensions, dependencies and infrastructure.

It also leads to better defense.

The Takeaway

Glassworm is not only a story about a botnet.

It is a reminder that the open-source supply chain is an investigation surface.

The developer is part of that surface. So is the extension marketplace. So is the package registry. So is the CI/CD pipeline. So is the command-and-control layer hidden behind legitimate or decentralized services.

For OSINT analysts, the task is to connect these layers without overstating certainty.

Start with the timeline. Separate account activity from code activity. Map the dependency chain. Preserve evidence. Treat public metadata as evidence, not decoration. And never assume that a trusted account means a trusted action.

In modern supply-chain incidents, the most important clue may not be the malware.

It may be the path that made the malware look ordinary.