Tutorial: Automating OSSEC HIDS Deployment on Modern Infrastructure Pipelines for Security at a Touch

The more cloud services grow in complexity, size and reach, the more security and automation need to be enforced. In this security overview and tutorial, we will take a look on how to secure our infrastructure by deploying OSSEC HIDS while circumventing its limitations to meet modern infrastructure automation requirements.
Part 1: Security Basics

In today's Internet landscape, security is a priority. Since security threats are extremely varied in nature (ranging from rootkits to DDoS attacks), there are many different tools to monitor, prevent, and mitigate them. In addition to other security mechanisms, such as firewalls and antivirus software, it is important to implement an **Intrusion Detection Systems (IDS)**.

**Intrusion Detection Systems (IDS)**

An **IDS** is a device or software application that monitors a network or systems for malicious activity or policy violations. Any detected activity or violation is typically reported either to an administrator or collected centrally to be audited.

There is a wide spectrum of **IDS**, ranging from antivirus software to hierarchical systems that monitor the traffic of an entire backbone network. The most common classifications are **Network Intrusion Detection Systems (NIDS)** and **Host Intrusion Detection Systems (HIDS)**.

**Network Intrusion Detection System (NIDS)**

A more traditional approach to dealing with intrusions is to implement a **Network Intrusion Detection System (NIDS)**, which basically probes for patterns on the network (such as excessive traffic, traffic on unknown ports or misbehaving traffic patterns) to identify security threats.

However, this approach has some limitations:

- **Configuration compatibility**
  To properly deploy a **NIDS**, the network architecture and security has to accommodate it (for instance, port spanning or packet mirroring), which may not be optimal nor feasible.

- **Performance**
  Depending on the scenario, mirroring and thoroughly analyzing packets can cause throughput bottlenecks and slow down the network as a whole.

- **Flexibility**
  Changing network topology and/or including new services inside the network is likely to require administrative intervention and upfront knowledge about applications and traffic.

- **Granularity**
  Threat detection can only go as deep as the network level. If there are threats that have low network surface, they can be harder to detect.
On more traditional and predictable on-premises static environments, this lack of flexibility, granularity and need for upfront knowledge could be overlooked.

However, in this age of **Infrastructure as a Service (IaS)** and **DevOps/CI** pipelines, infrastructure needs to grow or shrink on demand (elastic) and components are usually transient, disposable and easily rebuildable.

**Host Intrusion Detection System (HIDS)**

To secure this infrastructure in these new scenarios, an alternative approach to **NIDS** is to deploy a **Host Intrusion Detection System (HIDS)**.

**HIDS** deals with security threats inside host machines themselves (for instance privilege escalation, rootkits, logs, etc.), rather than on the network.

This has some clear advantages:

- **Configuration compatibility**
  
  **HIDS** needs little additional configuration to be accommodated on existing network infrastructure.

- **Performance**
  
  **HIDS**-specific protocols tend to be lightweight, having little impact on the network and on the host.

- **Flexibility**
  
  Since each host is individually monitored, adding/removing hosts and services is easier.

- **Granularity**
  
  Each host can have many different monitoring rules, which means that we can be as specific as we want on what to monitor for security threats.

In this tutorial, we take a closer look at an **OSSEC HIDS**, and how it can be deployed through automation to supplement missing functionality, in order to fulfill dynamic requirements.
Part 2: What is OSSEC HIDS?

OSSEC (Open Source Host-Based Intrusion Detection System) is an HIDS that monitors a wide assortment of events types that may indicate an invasion and matches these events to rules that, in turn, trigger responses.

It has two major components:

- **Manager** (server)
  The manager is the central piece of the OSSEC deployment. It stores the file integrity checking events. All the rules, decoders, and major configuration options are stored centrally in the manager; making it easy to administer even a large number of agents.

- **Agents** (clients)
  The agent is a small program, or collection of programs, installed on the systems to be monitored. The agent will collect information and forward it to the manager for analysis and correlation. Some information is collected in real time, others periodically. It has a very small memory and CPU footprint by default, not affecting the system’s usage.

These components match rules and trigger warnings and responses:

- **Rules**
  Events in the system can be virtually anything; OSSEC already comes with many built-in events, like: port knocking, failed authentication, too many connections from a host, USB insertion, logs generation and content, etc. Rules are used to catch these events on agents and send them the manager, which will generate warnings and trigger responses. Rules are extremely customizable and extendable, and are categorized in levels, from 0 (least severe) to 15 (most severe).

- **Warnings**
  Once a rule is triggered, it generates warnings. These warnings can then be forwarded via e-mail or integrated with other software (like slack or instant messengers).
- **Responses**
  After a **rule** is matched, then the monitor triggers the **response** on **agents**, such as denying a specific host, or stopping a given process, etc.

**OSSEC Limitations**

We are in a time where applications, infrastructure and continuous integration tools have reached such complexity that automation is not just a matter of good practice anymore, it is a necessity.

As infrastructure grows in size and complexity, we need to enforce infrastructure as code, so it can be easily reproducible, disposable, versionable and idempotent. To that end, security enforcement tools, such as **OSSEC** itself, are no exception.

While **OSSEC** excels at its scope, there are issues with its deployment when it comes to automation:

- **Interactivity**
  Standard procedures to manage **manager** and **agents** involves interactive prompts, which require (much) human interaction.

- **Security**
  Although there is some degree of automation, the current stable version 2.8.3 (at the time of this writing and deployment) doesn’t support authentication for **agent** self-inclusion, which could be a security liability.

- **Management**
  Most management operations require the **manager** to be restarted manually. There is no event handling for management operations whatsoever.

- **Idempotency**
  It is hard to keep track of both **manager** and **agents** through reinstalls. Since our automation should be idempotent, each fresh install or reinstall of the **manager** would destroy the current list of **agents**, leaving them orphaned. We need a way for the **agents** to check if the current install is sane and associated with the current **manager**, and if not, correct that.

Our goal is to circumvent these limitations in an automated and secure way, while keeping the **OSSEC** core functionality intact.

**Automated OSSEC**

A solution to circumnavigating the limitations of **OSSEC** is to automate it, through the use of our own infrastructure code.
Part 3: Tutorial

Overview for an Automated OSSEC Deployment

One of the biggest limitations to automate an OSSEC deployment is the need for administrative intervention even for basic tasks (such as adding/removing agents or restarting the manager).

To circumvent that, the following is required:

- **ossec-authd**
  This is the default way of adding agents without manual intervention. However, there is no authentication mechanism available on the latest binary available (2.8.3 at the time of this writing), which could be a security liability. To use ossec-authd with ssl, we need to install version 2.9 from source, in an automated way.

- **Agent-side event triggering**
  We need to create a way for the agents to trigger events on the manager. In our tutorial, we did this by creating listeners on the manager, and a basic file-based protocol.

- **Registration**
  An ossec-authd is ran on the manager, waiting for a registration request. On version 2.9 the registration is authenticated through pre-shared SSL keys and certificates. Once a connection is made and the SSL keys are matched, the agent is added to the manager.
• **Messages**
  These are plain-text files containing the agent **hostname** and **ID** (if the **agent** is not yet registered on the current **manager**, this value will be **NEW**). This file is uploaded to each **listener** depending on which event we are trying to trigger.

• **Listeners**
  These are SFTP jails with Linux inotify daemons waiting for a **message** file to be written. Once a **message** file is sent via SFTP to the directory monitored by the listeners, the relevant listener, indicated by the message file filename, parses the **message** file and triggers different actions. Also, each listener is running on a terminal multiplexer, and logs to both stdout and to a log file.

**With these mechanisms, we can automate many use cases that, otherwise, would need manual intervention.**

**System Requirements**

While striving to achieve this solution, we outline the following requirements, to make our solution robust, repeatable, and solid:

• **Use Ansible for automation**
  The infrastructure that we used for this tutorial already uses Ansible. There is no need to increase our stack’s complexity by changing configuration, since Ansible is powerful and consolidated enough to handle the project’s needs.

• **Use upstream code**
  **OSSEC** being open-source, means that we could fork the code, implement our extra functionality and issue a merge request. However, with this approach, we would need to maintain a separate codebase until the pull request gets merged (if ever), while backporting any eventual changes until then. Also, the sensitive nature of **OSSEC** scope compels us to use code that is already audited and tested through many different deploys, rather than experiment with custom code.

• **Simple and open dependencies**
  While implementing our custom deployment, we should strive to use open-source software and UNIX standard tools.

Because we are using **Ansible** for provisioning everything needed to automate our solution without intervention, the code needs to be organized in **playbooks** with different **tasks**.
Step 1: Create the Manager Playbook

For provisioning the manager, you will use a pretty straightforward playbook with a main.yml task and template files containing configurations, code and keys.

Step 1.1 Create the main.yml Task File

This should describe the following, in order:

- **Step 1.1.1: Install OSSEC dependencies**
  Install dependencies for OSSEC to work properly on the chosen Operating System (we used Ubuntu 14.04). In our case, it was SSL development libs, postfix server (for mail alerts) and some specific fixes for Ubuntu 14.04.

- **Step 1.1.2: Setup SFTP jail user and files**
  Create a user named **ossec-agent** which will own all the listener related files, including ssh keys needed. This is where the message files from the **agents** will be kept.

- **Step 1.1.3: Install OSSEC from source with authentication support**
  ossec-authd SSL authentication is only available on version **2.9**, which is only available through source, rather than binary packages. So, install it from source with some previous
configuration templates. Also, it needs to use templates for basic manager configuration and slack integration.

- **Step 1.1.4: Start the listeners**
  Idempotently start listeners which are basically notify daemons running on screen pseudo-terminals; and ossec-authd. This is done by killing any screens that may have been created from a previous run and starting them again.

**Step 1.2: Create the Listeners**

- **Step 1.2.1: Create start_trigger.sh**
  Starts OSSEC manager and forces a syscheck on the agent that triggered it. It waits for a message file to be written/updated and parses it to get the agent id and name. Afterwards, it updates the client.keys file, uses the id and name to run a system check and write to logs and stdout, to identify which agent triggered these events.

- **Step 1.2.2: Create stop_trigger.sh**
  Stops OSSEC manager. This is used to avoid polluting monitoring alerts during provisioning, since an ansible run would give many false positives.

- **Step 1.2.3: Create restart_trigger.sh**
  Restarts OSSEC manager. This is a shortcut to stop, and then start, but without forcing a system check on the agent.

**Step 2: Create the Agent Playbook**

The playbook for agents is a little more complex in structure. It is a shared role that can be provisioned on top of another arbitrary playbook (for instance, a web server playbook). When using this shared role from another playbook, we must set a value for the action variable, that will then be used for the agent shared role playbook to decide which set of tasks will be ran on provisioning.

The action variable can have the following values:

- **stop**
  Triggers manager to stop. This is used to avoid unnecessary warnings while provisioning other playbooks.

- **Install**
  Installs agent on the server being provisioned.
- **start**

  Triggers **manager** start and forces a system check on the server being provisioned.

  ![Diagram showing the playbook workflow]

  The playbook will use the **action** variable together with the **check_registration** task to manage different situations. It has the following **tasks** files:

**Step 2.1: Create main.yml**

The **main.yml** task file will probe for the agent IP, check if the agent is already registered on the current **manager**, and decide which tasks will be run. It could be one of: **install.yml**, **stop.yml** or **start.yml**.

**Step 2.2: Create check_registration.yml**

This task is used to check whether the server being provisioned is already registered on the current **manager**. This is necessary because the **manager** could be reprovisioned, which would scrap all information about previous **agents** (More on this in the **Part 4: How it Works in Practice** section). With this task, you can check if that's the case and set the **is_registered** boolean variable that will be used on other tasks to take appropriate measures (like re-register, or ignore events).

This is done by comparing the current **manager** agents list (on **client.keys** file) with the **agent ID** on that list. If this is not the same, it means that the **agent** has an outdated install.
A **client.keys** file is generated and updated by the **manager**, which keeps track of the current **agents** registered. It has the following anatomy: `<ID> <name> <ip> <hash key>`

Each **agent** registered on the **manager** will have an entry like this on the **client.keys**, and each **agent** has a copy of it containing only its own entry.

What **check_registration.yml** does is to check if the **client.keys** entry on the **agent** has is contained on the **manager** **client.keys** file. If it does, it means that the **agent** is already registered on the **manager**.

**Step 2.3: Create stop.yml**

This task send a message to the **stop listener** on the **manager** and stops the ossec **agent** itself.

**Step 2.4: Create install.yml**

Similar to the **main.yml** task from the **manager** playbook, this installs the **agent** and sets up all files needed for messaging the **manager**.

After installation and registration, trigger files are updated using the **agent_info** file. This is needed because it may have changed, in case the **manager** was reprovisioned. If this file is empty, or cannot be found, this means that the **agent** is yet to be registered (**is_registered** variable is false), so the trigger files will be populated with **NEW**, instead of the ID.

**Step 2.5: Create start.yml**

This task starts the **agent**, sends a message to the start **listener** and updates the **message** files.
Part 4: How it Works in Practice

With these mechanisms in place, we can take a look on how operations are done through automation:

Installing the Manager

With the **manager** playbook, many installation steps are automated such as dependencies installation, source downloading and building, **listeners** and authentication daemon (**ossec-authd**) setup.

Installing and Adding New Agents

Similarly with the **manager** provisioning, the **agent** playbook automates many installation steps. After installing the **agent**, it registers itself into **ossec-authd**, and then sends a **message** to the **start listener**, so it will restart the **manager** to complete the registration and force a system check on the **agent**.
Reinstalling an Agent

If we reprovision a server that is already registered as an agent on the manager, it will send a message to the stop listener, so to avoid polluting with unnecessary warnings. Afterwards, it will skip registration (to avoid duplicated entries) and sends a message to the start listener, to resume operation, while keeping the connection.

Reinstalling the Manager

If we reprovision the manager, all the current agents information list is lost. After that all the agents must be reinstalled, where the agent playbook will detect that the current connection list (client.keys) is outdated, so it will re-register on ossec-authd and send a message to the start listener.
Part 5: Conclusion

Security is a vital piece of any modern infrastructure, and although an HIDS such as OSSEC is a great tool to enforce stricter security directives, it fails to comply with important DevOps/CI pipeline requirements. Using modern automation and DevOps techniques and tools, we are able to circumvent such limitations and adapt the solution to specific needs and environments.
iRonin prides ourselves on being DevOps experts, who take security very seriously. We have deployed many security systems for our clients and love a challenge. We create solutions that fit our clients needs, no matter the complexity. Contact our offices for more information about how we can help your business achieve run more securely, efficiently, and easily.

Get in touch

www.ironin.it  
contact@ironin.it  

North America: +1 (385) 282 4499  
Europe: +48 (22) 397 8866  

©2017 by iRonin  |  Web Mobile DevOps Experts  |  www.ironin.it