VB2020
localhost

30 September - 2 October, 2020 / vblocalhost.com

LIKE BEES TO A HONEYPOT – A JOURNEY THROUGH HONEYPOT DEPLOYMENTS

Matthias Meidinger
VMRay, Germany

matthias@meidinger.de

ABSTRACT

Honeypots can provide valuable insights into the threat landscape, both in the open Internet as well as on your internal network. Deploying them correctly is not always straightforward, and neither is interpreting any activity on them. This paper aims to convey the necessary knowledge to start deploying a honeypot infrastructure and benefit from it. It highlights the considerations and pitfalls that can be encountered in the deployment of honeypots and the supporting infrastructure. Furthermore, the paper showcases automation, aggregation and visualization of honeypot activity based on a production deployment.

INTRODUCTION

As attacks on Internet-facing infrastructure shifted to being mostly automated in recent years, honeypots lost some of their meaning in detecting novel exploits and attacks on said infrastructure. Combined with the fact that the people running honeypots usually don't want to give away details on how they have customized them to keep them from being detected, this has led to a situation where the value of running them has been upstaged. However, although the means of operation of attackers has changed, honeypots still provide valuable insights into ongoing campaigns, the credentials used, and the distributed payloads.

As an introduction to the reader, the definition of a honeypot that we base the remainder of the paper on is as follows:

> 'A honeypot is a (…) system intended to mimic likely targets of cyberattacks.' [1]

Based on this definition, it can be concluded that honeypots usually mimic systems that look vulnerable and are therefore valuable targets for attacks. The systems mimicked can be either vulnerable-looking services (e.g. SSH, Elastic) or clients (e.g. browsers). In the latter case, a browser is emulated to find websites that, for example, try to execute malicious payloads on clients, like JavaScript cryptominers or drive-by downloads.

Meanwhile, in the former case the honeypot emulates a complete server or protocol to find tools, techniques and procedures used by malicious actors. Such honeypots can be used to uncover, for example, attacks tailored to overtake publicly accessible IoT devices or to ransom unsecured MongoDB instances.

Server-side honeypots can further be grouped into three categories based on the level of emulation they provide: low, medium and high interaction honeypots.

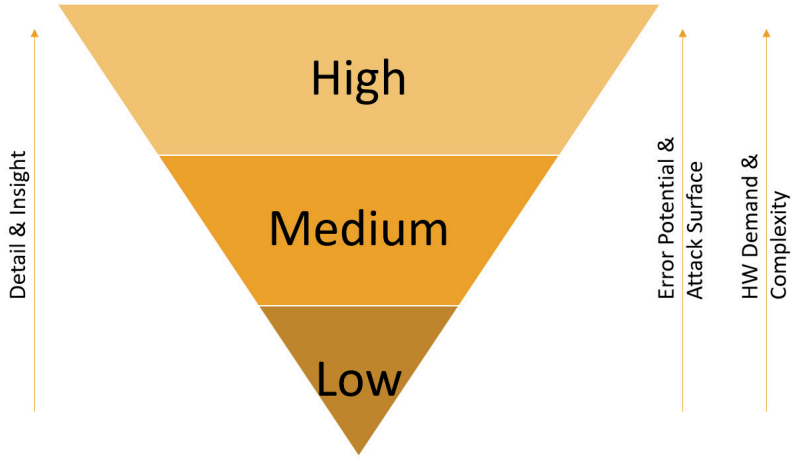

Figure 1: Server-side honeypots can be grouped into three categories based on the level of emulation they provide: low, medium and high interaction honeypots.

Low interaction honeypots are rather easy to build, as they often emulate only the basic commands of a protocol. For SSH, a low interaction honeypot can consist only of the login dialog to collect usernames and passwords potentially used in credential stuffing attacks.

Medium interaction honeypots take this principle a step further and emulate more commands and part of the surrounding system. As an example, the medium interaction *Cowrie* honeypot [2] emulates a complete filesystem as well as many integrated system commands like lsof or netstat to look like a fully running system.

Finally, high interaction honeypots represent a fully functioning implementation of the protocol in question, often made available through a man-in-the-middle (MitM) proxy which logs every interaction with the honeypot. For SSH this is represented by *Dockpot* [3], which is a honeypot that runs a full *Linux* system in an image, exposing the SSH connection through a MitM proxy that logs all interactions and issued commands. For every connection from a distinct source IP, a new container will be created and kept until a timeout is reached. This not only enables connection separation but also

persistence across connections, as the attacker finds the filesystem with all changes and additions that were conducted during the first connection.

All three groups have their advantages and use cases. While the level of detail and insight grow from low to high interaction honeypots, the error potential, attack surface, hardware demand and general complexity increase as well.

Low and medium interaction honeypots are often developed as scripts being run by an interpreter, e.g. Python. While they provide limited insight and are relatively easy to detect, they can be installed on virtually any OS that is able to run a fitting Python distribution. This could be anything ranging from a *Raspberry Pi* to fully fledged standalone hardware or cloud deployments.

High interaction honeypots are often based on virtualization or containerization technologies and require a more advanced setup. This includes using sufficiently powerful hardware, configuring the abstraction layer, and setting up VMs or containers. Therefore, goals, budget and time constraints should be known before deciding which honeypot to deploy. The remainder of this paper details precautions and pitfalls for the deployment of honeypots, ways to efficiently collect the produced data, sighting and visualizing this data, as well as examples for automation of workflows and a closing conclusion.

DEPLOYING HONEYPOTS

The first step to data collection, which is also the most important, is the deployment of honeypots. There are multiple pitfalls and recommendations to consider depending on the use case. After a successful deployment, the next step is to collect the generated data and possible payloads at a single data sink to enable the generation of metrics and monitoring of the complete infrastructure.

Deployment considerations

There are two main scenarios to consider when deploying honeypots: internal versus Internet-facing deployments. Both are valid scenarios but they cover different use cases. For the remainder of this paper, we focus on Internet-facing deployments for data collection if not further defined.

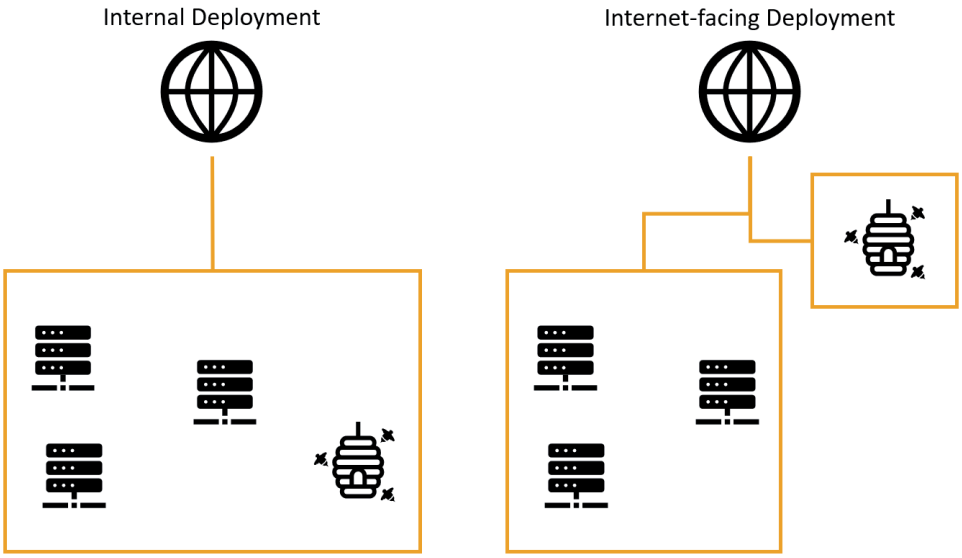

Figure 2: Internal versus Internet-facing deployments.

In an internal deployment, honeypots can be considered as traps or alert systems. The idea is to deploy them throughout the company infrastructure, preferably near production servers. If an attacker is looking for a foothold in a network, they stumble upon these strategically placed systems and try to use them to persist access. Ideally, these honeypots will have been set up to raise alarms if incoming connections are detected, as there is no legitimate use for them in daily operations. This scenario can support existing measures like intrusion detection systems or log monitoring as an active component to increase the chances of early detection of intruders.

Internet-facing deployments, on the other hand, are more tailored towards collecting data on widespread attacks. This can range from basic information such as attacked services (e.g. how common are attacks against *Android Debug Bridges*) or credentials used, up to detailed TTP information (i.e. which commands/scripts are executed, attempted lateral movement, persistence techniques and possible evasion attempts). In contrast to internal deployments, these are constantly exposed to world-wide traffic. Therefore, they are always to be considered compromised. As these deployments aim to provide no direct protection to an internal network, it is advisable to isolate Internet-facing honeypots completely from the production infrastructure.

Besides these specifics, we can also derive some general recommendations for all deployment scenarios.

As these systems are considered insecure by design, it is advisable to treat them accordingly. Leaving production data or company information on them is inadvisable, as is reusing usernames, passwords, certificates and SSH keys. Otherwise, if attackers manage to escape from the honeypot to the hosting OS, they will be able to gain valuable information about internal infrastructure and active usernames.

Furthermore, it is strongly advised that honeypot services should be run as a non-root user that has minimal permissions and is not able to use sudo. In the case of honeypot escapes this makes it considerably harder for attackers to escalate privileges. As most emulated services are running in the range of system ports which require elevated privileges, it is prudent to run them on non-system ports and utilize iptables forwarding rules to make them look like they are running on the system port. If honeypots for common services like SSH and FTP are deployed, they should be running on the services' default port. Especially for SSH as means of access for most systems, it is recommended to disable password authentication and root login for the real SSH server, as well as running it on a non-standard port to free up port 22 for the honeypot. This also means that the creation of an SSH alias in local configs is recommended to avoid connecting to the SSH honeypot by accident when conducting maintenance or applying configuration changes.

Another consideration is the hosting service for the infrastructure. If it is not hosted on company-owned infrastructure, the idea of using a low-end VPS provider is compelling. Unfortunately, these are prone to being shut down in the context of deadpooling scams [4], so it pays to be prepared to lose these systems at any time. In general, automated deployments based on tools like *Ansible* or *Puppet* should be used for reproducible results and to lower the risk of misconfigurations. Combined with a backup strategy for collected data, logs and payloads, this ensures resilience to data loss.

Furthermore, it is recommended to minimize the usage of OS-based resources for the specific requirements of honeypots. For example, the usage of local virtual environments for Python-based projects should be considered over using system-wide package installations to avoid dependency problems with multiple projects running on the same language or OS updates that break dependencies.

Regarding operation, it is also advisable to monitor the regular operation of deployed honeypots including storage utilization, ideally with automated tests tailored to the respective protocol.

Generally speaking, you're exposing a system to the world that looks vulnerable – it most likely *is* vulnerable, but in other ways than you'd think. Honeypot deployments, especially when Internet facing, are an asymmetrical playing field with an attacker advantage. Attackers have infinite ways and time to try attacks – the operator only needs to make one mistake to expose the honeypot host and possibly the surrounding network to attacks.

Besides the deployment, there are more things to take into consideration. Attackers constantly try to detect honeypots – with various techniques and varying success rates. A talk detailing finding flaws and their implications was held at 32c3 [5]. In the next section, some commonly encountered detection techniques and possible workarounds are presented. These are merely pointers in the right direction. It is advisable to monitor your honeypot infrastructure constantly and keep an eye out for disconnects always happening after specific commands or workflows, as these can point to evasion strategies.

Custom configurations

Many honeypots come with a default set of emulated parameters including hostname, service version and credentials. This is especially common in low and medium interaction honeypots. In the context of honeypot configurations, customization is the key to evasion mitigation.

As an example, the SSH honeypot *Cowrie* is considered. If the default configuration is not changed, it used to accept the user 'Richard' with the password 'fout', afterwards announcing that its system name is 'svr04'. Checking for default configurations like these is relatively easy and therefore happens quite a lot.

As a preventive measure, the footprint of the honeypot should be as customized as possible. In particular announced hostnames, service versions and banners are low hanging fruit that can be changed. For low and medium interaction honeypots it can also be a valid strategy to change the outputs of emulated commands and create custom filesystem pickles to further make the system unique.

Finding evasion tactics

As a general recommendation, you should monitor your honeypots closely, especially in the early days of deployment, as they are 'fresh' and unknown at this point. To stay with the example of *Cowrie*, it is possible to spot evasion techniques quite easily in the generated logs. In all cases the command workflow on the system is the same up until a specific point where commands either fail or are not executed at all. A commonly observed pattern of actors on the *Cowrie* SSH honeypot is to echo the raw script into a file and subsequently try to execute it.

```
user@pot:~$ /bin/busybox echo -en '\x00\x00\x00\x00\xb4\x03\x00\x00\x1e\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00' >> retrieve
user@pot:~$ ./retrieve
```

This does not work on most low and medium interaction honeypots, as they disown files created by the connected SSH user as soon as possible. As these files are now non-existent, the workflow of echoing the initial payload into a file and executing said file afterwards does not work on these honeypots, therefore it can be considered a successful evasion.

Another evasion technique commonly encountered is to download the payload and execute it in a second command. As discussed, this approach leads to a successful evasion, as the file is no longer available to the user at the time of execution.

```
user@pot:~$ cd /; wget http://45.148.10.175/bins.sh; chmod 777 bins.sh; sh bins.sh;
```

There are not many mitigations available for these issues. Due to the very nature of low and medium interaction honeypots, the most viable mitigation is to switch to a high interaction system. High interaction honeypots present a complete, persistent environment to an incoming connection that is often cached even through reconnects. This means that all dropped or downloaded payloads are available for execution instead of being snatched away by the honeypot.

Sanity checks

Sanity checks are also encountered quite often. As an initial example an SMTP honeypot is considered. Attackers will try to connect to a mail server and send a test mail to their own infrastructure to check if the mail server is allowing outbound mail traffic.

```
{
  "timestamp": "2020-06-14T08:57:00.854855",
  "src_ip": "0.0.0.0", "src_port": 54282, "eventid": "mailhon.data",
  "envelope_from": "spam@provider.com", "envelope_to": ["spam@provider.com "],
  "envelope_data": "From: spam@provider.com\r\nSubject: 42.42.42.42\r\n
     To: spam@provider.com\r\nDate: Sat, 13 Jun 2020 23:56:59 -0700\r\nX-Priority: 3\r\n"
}
```

A possible mitigation is to allow the first mail from every connection to leave the honeypot. Be advised that this bears legal implications as, technically, the system is sending out spam.

Besides the fully-fledged production test there are other sanity checks that can be observed. As a general guideline, deployed honeypots should expose a configuration and sizing that is similar to their real-world counterparts. This can be achieved more easily on low and medium interaction honeypots as they often emulate commands by looking up text files which massively eases the spoofing of cluster states, replica configurations, or even filesystem sizes.

SIGHTING DATA

After successful installation and customization, the deployed honeypots start generating data. One of the main challenges at this stage is sighting the logs and finding interesting events.

As logs are not tailored for human consumption, they are notoriously hard to read and check. This is where visualizations come into play. The author recommends ingesting logs into a central system like *Elastic* or *Splunk* that indexes the generated data.

Besides making all log data available for dashboards, there is the added advantage of making logs of all deployed honeypots of your whole infrastructure available on a central system. This enables dashboard and report generation across the whole infrastructure and deeper insights.

For the remainder of this paper it is assumed that all logs are collected in a central *Splunk* instance, which is also used for the shown dashboards.

Some key metrics the author finds useful for daily work are:

- Connecting source IP
- Number of different source IPs in the last 60 minutes

- Top 10 connection counts by source IP
- Username / password pairs (failed & successful)
- SHA256 hashes of captured payloads
- List of executed commands (depending on honeypot)
- Unique connection identifiers (i.e. SSH keys or client version strings)

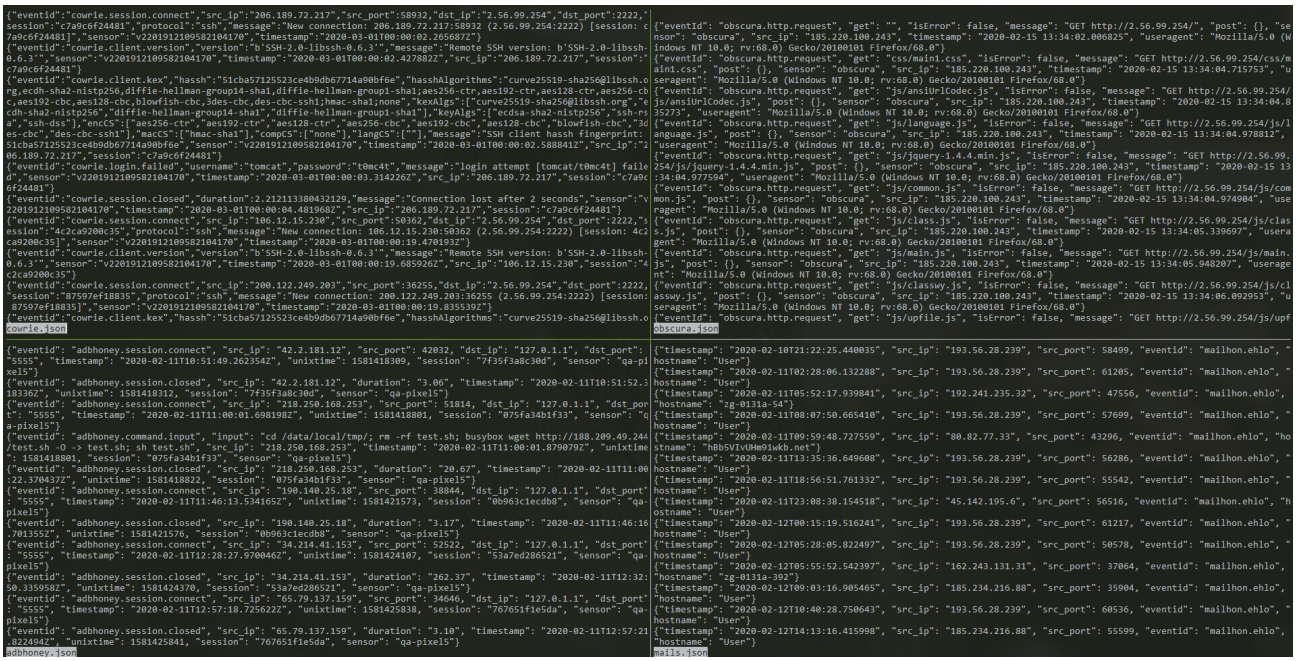

Figure 3: The raw logs are not tailored for human consumption.

Using dashboards is no turnkey solution to better insights into produced log data. It is still the responsibility of the user to clean up data first before plotting it. In this context it is viable to consider lending common dataset preparation procedures from the area of machine learning [6]. Some pointers can be the need for deduplication of events, e.g. multiple connections from the same source IP. Whilst in some scenarios this can be interesting, e.g. to find credential stuffing attacks, it can be counterproductive in others, such as the absolute count of unique connections in a given timeframe.

Source	# times seen	SHA256 Hash	Target Filename	User	Password
stdin (Telnet)	35	720171859a1a2b97cb90a666395d1678faf62496999c267f6e7b73830be5aa8b	/tmp/up.txt	pi	password123
stdin (Telnet)	16	a5097c8a4ac74f075940ed707587d165650c31549a76220ce7d0afad8b54468c	/tmp/up.txt	service	service123
stdin (Telnet)	29	40f1e75930654a56e1f2e10c9e9ff525f5d23e3e9941f7de858e5860e30a5d4	/tmp/up.txt	pi	12345678
stdin (Telnet)	1	06c5240aa9c4ac7aead15a526c1ba3d7fcffa67bfc0578b051c50c5bd146dfc6	/tmp/up.txt	service	1234
stdin (Telnet)	1	64ff5277ccfe4a85ec229ef058ec85bdb3ca49261121b47491511c929e40bb64	/tmp/up.txt	service	service
stdin (Telnet)	1	781bd7066deb6077021812917ec0c6046127225eaf6b0d59a7d638e8a570cb3d	/tmp/up.txt	pi	pi2020
stdin (Telnet)	2	d3a7af7e678c5b3ff0928216204d32b6c8007ec8e3b45529246f8bbe803a779c	/tmp/up.txt	service	123
stdin (Telnet)	1	17d8840eccfd52ca260e03cff72f770cc14ce6270e192cd5767021a56b419a2	/tmp/up.txt	pi	pi2019
stdin (Telnet)	1	57b5d04212af720307e076bc73110f3bd8389383a72847f07a42b5b4a2d6592d	/tmp/up.txt	service	password123
stdin (Telnet)	1	2718fda345ed5591c16418 6b24486109e6e48a2f89142a123c58fa9b64d01169	/tmp/up.txt	service	password
stdin (Telnet)	1	77b9259791988c48079801ef0fc107fb8f042133e1671c0e995800362614aeef	/tmp/up.txt	pi	1234567

Figure 4: An example of noise in a dashboard.

The screenshot in Figure 4 shows an example of noise in a dashboard. The file '/tmp/up.txt' is generated with different content but is always written to the same path. While itself part of an evasion technique, it also fills up dashboards that show the latest payloads found. This is where filtering can help to keep dashboards effective by lowering noise. After validating that the created file is indeed noise, it can be filtered by its path. Nevertheless, both the contents of this file and its importance might change. With the filter in place, this change might easily be overlooked. Therefore, continuous data analysis and pattern recognition are required to keep a dashboard valuable and usable.

AUTOMATING WORKFLOWS

At this point the infrastructure consists of several running honeypots that are producing data, which in turn is sent to a *Splunk* instance for indexing and dashboard generation.

Having a static dashboard with basic metrics is helpful for getting a grasp on the state of the infrastructure. If honeypots are used in a more active scenario, i.e. for threat hunting, it is favourable to add common lookups and shortcuts to a dashboard

to improve initial triage times. The proposed *Splunk* dashboard therefore contains contextual links to *VirusTotal*, *Shodan* and *GreyNoise*.

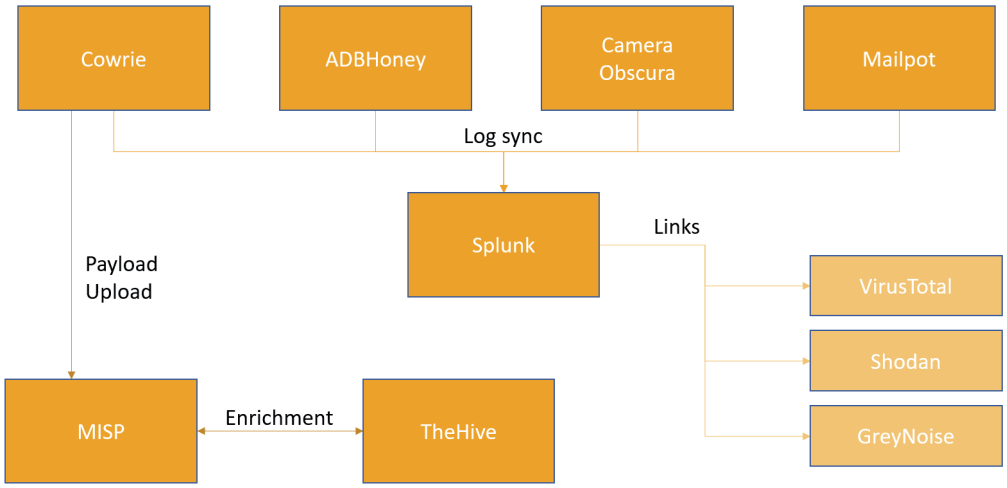

Figure 5: Workflow.

All encountered SHA256 hashes are direct links to *VirusTotal* searches, clickable IP addresses refer to *Shodan*, and Autonomous System Numbers (ASNs) are used as a lookup for *GreyNoise*. These services should provide enough information to decide whether a detailed investigation could lead to interesting insights.

To further decrease the number of manual tasks, one can also consider the usage of threat intelligence platforms (TIPs) like *MISP*, which offer automated enrichment and analysis capabilities for submitted samples. Most honeypots either already have API capabilities to upload payloads to a target server or can be retrofitted to do so with little effort. In the showcased infrastructure, *Cowrie* is configured to query a *MISP* instance for the SHA256 hash of every encountered payload. If the payload is unknown, a new case is created in *MISP* and the payload is attached to it. If it was encountered before, a 'sighting' event is added to the according case.

The advantage of platforms like *MISP* is the community aspect and the integrated enrichment capabilities that can give samples and payloads context, IOCs, and analyses of other members of a sharing group. In the presented architecture, this role is fulfilled by a tandem of *MISP* and *TheHive*. *TheHive* is another TIP that focuses more on external integrations and analysers. In its current state, every encountered payload is uploaded to *MISP*, followed by an automated case creation in *TheHive*. This enables analysts to run analysers with little additional overhead, as they do not need to create case files and upload samples by themselves.

This area of the proposed architecture can also be carried out by a security orchestration, automated response (SOAR) system to further automate responses and increase analytic capabilities.

Example workflow

To illustrate the described system integration and workflows, we assume a file was uploaded to one of the *Cowrie* SSH honeypots with the SHA256 hash of 69787a2a5d2e29e44ca372a6c6de09d52ae2ae46d4dda18bf0ee81c07e6e921d. As a first measure of interest, this file can be investigated in the *Splunk* dashboard.

The dashboard already provides some valuable information on first sight. It can be derived that the payload was uploaded using the default credentials for a *Raspberry Pi* and the connecting address was located to Switzerland. By clicking on the hash or the IP address, either *VirusTotal* or *Shodan* can be checked for initial information. Last but not least, a click on the ASN leads to a *GreyNoise* query that lists all known systems in this ASN. This can add context to the IP as it gives pointers if the ASN is known for malicious traffic.

After this cursory glance it is decided to further investigate this sample. Based on the information provided by *Shodan* and *VirusTotal*, the current working hypothesis is that this is a Bash IRC bot distributed by a system with a *Raspberry PI* SSH version string.

As the payload was dropped on an integrated SSH honeypot, it has already been uploaded to a connected *MISP* instance where a new case has been created (or, if the payload already exists, a sighting has been added). The event already has the uploaded file attached as a malware sample including some additional metadata like common file hashes, the size in bytes and the entropy. From here on, it is possible to make use of the *MISP* ecosystem to share and enrich encountered samples, for example through *MISP Communities* or *MISP Community Feeds*, as well as *MISP* plug-ins that integrate it with other products.

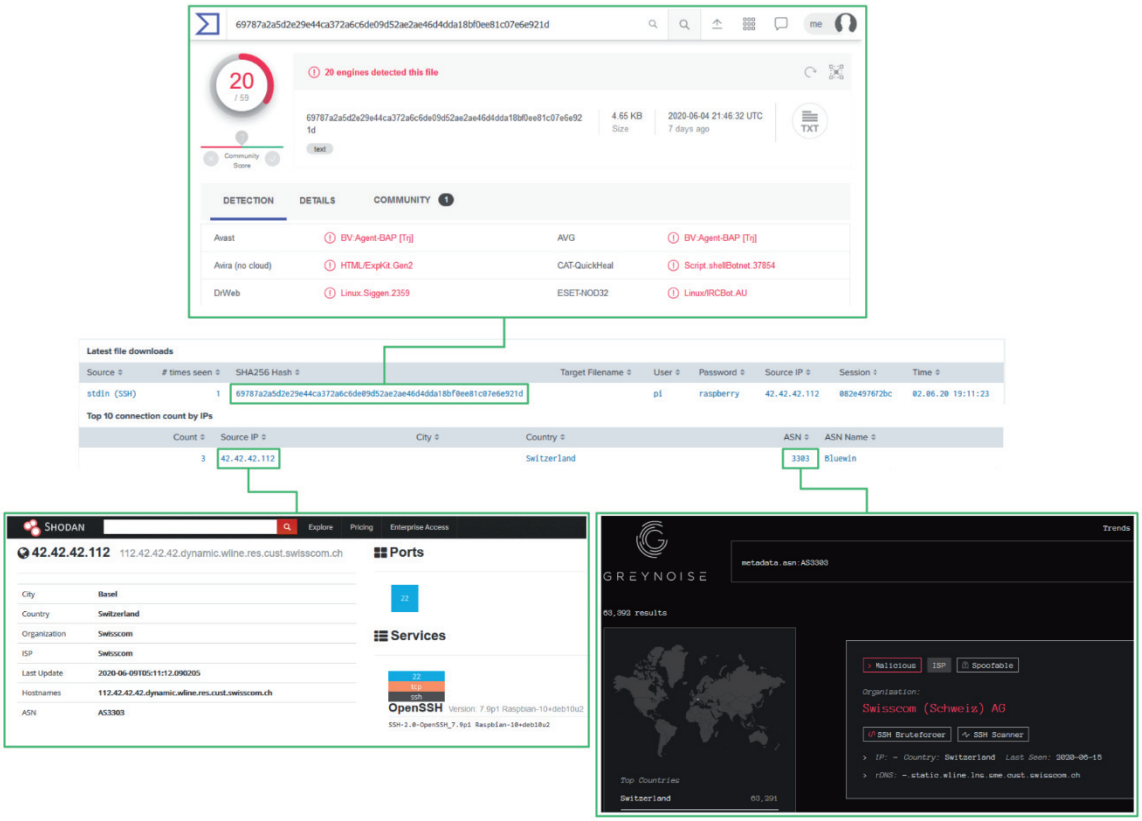

Figure 6: Investigation of file in Splunk dashboard.

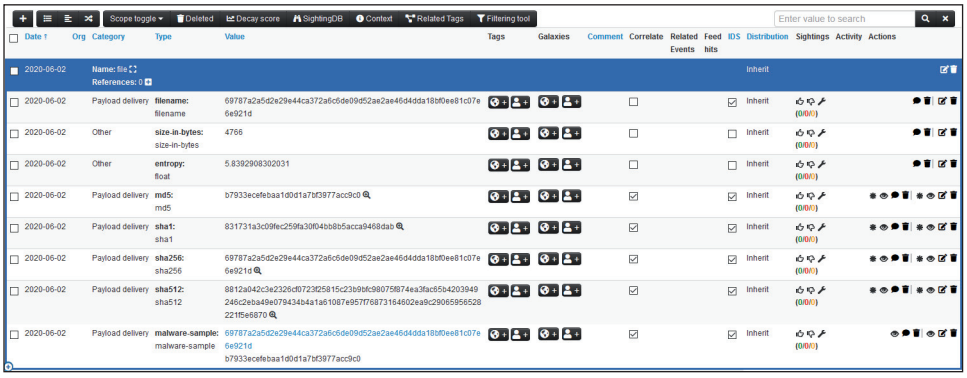

Figure 7: The payload has been uploaded to a connected MISP instance.

While the community aspect of *MISP* is its strong suit, there are other contenders regarding the effective use of integrations with third-party products. The tandem of *TheHive* and *Cortex* is an alternative that focuses more on said integrations. It consists of one or multiple *Cortex* instances that are responsible for running so-called analysers which are making use of several external services like *IBM X-Force*, *RiskIQ Passivetotal*, or *HaveIbeenpwned*. This is complemented by *TheHive*, which in turn offers case management, intel collection and templating capabilities.

Additionally, *MISP* and *TheHive* can work in two-way-synchronization mode, which unites the strengths of both platforms into an excellent solution for managing, tracking and optimizing investigations. For the example at hand this means that an incoming alert for the discovered IRC bot is awaiting its import as a case in *TheHive*.

The payload and all its observables from *MISP* are imported and available for use in *Cortex* analysers. As these are run, they generate additional observables and reports that can be added to the case at hand, as can be seen in Figure 8. The red tags attached to the hash and the file stem from critical results obtained by querying *IBM X-Force* and *VirusTotal*. All added metadata can also be synced back to *MISP* for integrity and sharing purposes.

At this point, an upload has been found and, without opening the file itself, a preliminary examination has been conducted, leading to the decision to further investigate the incident. The file was added to *MISP* and *TheHive* with minimal to no user

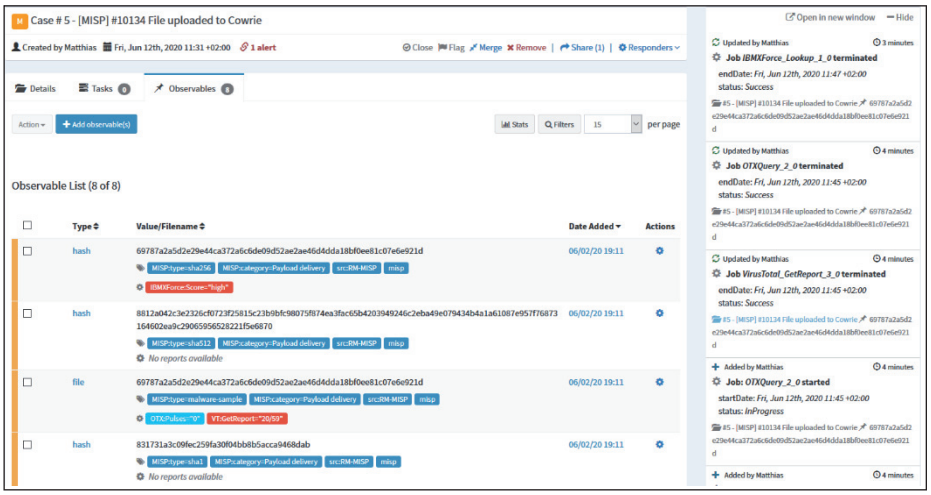

Figure 8: Additional observables and reports are generated that can be added to the case at hand.

interaction and made available to enrichment plug-ins and communities, therefore accelerating and improving the process of manual analysis and investigation.

CONCLUSIONS

Successfully deploying and integrating honeypots and supporting software into an existing infrastructure can be a daunting task that requires a decent amount of planning. Nevertheless, the advantages are evident:

If integrated correctly, honeypots enable faster alerting and a pre-emptive view into current attack strategies and automated attacks against publicly available infrastructure, whilst supporting integrations based on TIPs like *MISP* or *TheHive* speed up and improve the quality of triaging and lower the amount of manual work done by analysts. Combined with widespread log collection and well-designed dashboards, this compliments better defensive strategies and measures against novel attacks.

Especially with the continuous popularity of container-based virtualization technologies, high-interaction honeypots are expected to gain popularity and development traction. As it stands, this type of honeypot is considerably harder to detect, which makes it ideal for usage in Internet-facing deployments. This is due to the fact that the architecture mitigates most of the commonly used evasion techniques simply by being a fully customized system that behaves consistently and as close to a real system as possible.

Once deployed, honeypots are a low maintenance asset that can bring high value to the table, be it as a pre-emptive alerting system for internal infrastructure or as a sensor for discovering ongoing campaigns and credential stuffing attacks, collecting value intelligence without manual interaction.

As a closing note, some resources can be recommended for getting started with custom deployments. A good overview of common resources and projects is the repo *awesome-honeypots* [7], which can be of great service if a specific service or system is needed.

First and foremost, an all-in-one solution that bundles multiple honeypots with an *Elastic* stack, custom dashboards and a multitude of tools exists that is named *T-Pot* [8]. This project is developed by *Telekom Security* and offers a quick start for the price of customization. As the project is rather complex and relies heavily on containers, customization of the bundled honeypots is not as straightforward as it is in custom deployments. Nevertheless, it is an excellent starting point to get a feeling for deployments.

A step closer to fully customized honeypots are frameworks that abstract shared functionality between specific implementations. Examples are *DutchSec*'s *honeytrap* [9] and *Cymetria*'s *honeycomb* [10]. Frameworks can speed up the development process of custom honeypots but come with the price of predefined structure, as frameworks rely heavily on conventions to work correctly.

With the release of this paper, the presented *Splunk* dashboards are made available for general use and can be found in this repository [11]. This organization also holds repositories with the custom developed SMTP honeypot *mailhon* [12] as well as an IP camera honeypot, *CameraObscura* [13]. Finally, the last project that is used in the demonstrated environment is an *Android Debug Bridge* honeypot by the name of *ADBHoney* [14].

Last but not least, *Honeynet* [15] has to be named as a central research organization that is dedicated to continued development of honeypots as well as investigations into ongoing attacks.

REFERENCES

[1] What is a honeypot? How it can lure cyberattackers. Norton. https://us.norton.com/internetsecurity-iot-what-is-a-honeypot.html.

[2] Cowrie SSH/Telnet Honeypot. https://github.com/cowrie/cowrie.

[3] Dockpot. https://github.com/eg-cert/dockpot.

[4] 20 Low-End VPS Providers Suddenly Shutting Down In a 'Deadpooling' Scam. https://tech.slashdot.org/story/19/12/08/1549222/20-low-end-vps-providers-suddenly-shutting-down-in-a-deadpooling-scam.

[5] DeanSysman; Evron, G.; Sher, I. Breaking Honeypots for Fun and Profit. https://media.ccc.de/v/32c3-7277-breaking_honeypots_for_fun_and_profit.

[6] Six Steps to Master Machine Learning with Data Preparation. KD nuggets. https://www.kdnuggets.com/2018/12/six-steps-master-machine-learning-data-preparation.html.

[7] An awesome list of honeypot resources. https://github.com/paralax/awesome-honeypots.

[8] T-Pot – The All In One Honeypot Platform. https://github.com/dtag-dev-sec/tpotce.

[9] Honeytrap. Advanced Honeypot framework. https://github.com/honeytrap/honeytrap.

[10] Cymmetria honeycomb. An extensible honeypot framework. https://github.com/Cymmetria/honeycomb.

[11] Splunk Dashboards for your Honeypot infrastructure. https://github.com/CMSecurity/splunk-hp-dashs.

[12] CMSecurity mailhon. A Python3 SMTP honeypot. https://github.com/CMSecurity/mailhon.

[13] CMSecurity CameraObscura. IP Cam Honeypot. https://github.com/CMSecurity/CameraObscura.

[14] ADBHoney. Low interaction honeypot designed for Android Debug Bridge over TCP/IP. https://github.com/huuck/ADBHoney.

[15] Honeynet. https://www.honeynet.org/.