Breach Detection at Scale with AWS Honey Tokens

Daniel Bourke
Sr. Security Analyst, Atlassian

Daniel Grzelak
Head of Security, Atlassian

Honey tokens, by which we mean credentials or database records or DNS entries that set off alarms if you look at them funny, are extremely helpful for securing your enterprise.

We'll go over some infrastructure we've built to help deploy a specific type of honey token (AWS credentials) at scale (i.e. in a reasonably automatable fashion), as well as some things we learned by *accidentally* leaking a bunch of AWS credentials all over the internet.

I. What is a honeytoken?

We're using 'honey token' in this paper as a stand-in for anything you can lock down and fire alerts from. This can be nearly anything, depending on your context and capabilities: In a database, a record that won't get returned in normal business queries, but will get returned by an unwary attacker running `'SELECT * FROM IMPORTANT_TABLE;'` can be a honey token, as long as you alert if that record is ever queried. If you control a DNS server, you can set up alarms on certain subdomains being resolved, and sprinkle links to them in your documentation, where your employees will never see it but a curious interloper will spider it. Alternatively you might put some bogus internal email addresses in your CMS and if they ever start getting spam, you know someone's been peeking at your stuff. All of these are relatively easy to create for one-off or low-scale deployments, and you should consider doing so (or using a freely-available third-party service to do it for you).

II. AWS keys as honeytokens

AWS keys make extremely good honey tokens, because they're very interesting to attackers (because if you find someone's AWS keys, you may have just found several thousand dollars worth of cryptocurrency mining hardware in someone else's cloud); and because you, the defender, can really easily secure AWS keys, and alert if anyone tries to use them. They also have the convenient properly of being found in an enormous variety of locations, from developers' desktops to server environment variables to three months deep in your chat history from that time you just really needed to get that thing deployed.

AWS keys are great for this purpose, but it is a hassle to generate new keys and set up alerting for every time you deploy a new microservice, or add a new laptop to your workstation fleet. Or maybe you just want something to alert you whenever one of your 'private' repositories becomes public. If you want to do something like that, you'd need some kind of scalable, responsive infrastructure, with a customizable alerting pipeline and a sensible and easy-to-use API.

While we were waiting for that product to come to market, we wrote SPACECRAB. SPACECRAB is something you can deploy in an hour or so, which will provide you with an API endpoint you can use to create, update and dispose of AWS credentials, and a plethora (two) of alerting options (it's email or PagerDuty, but you can write your own as well).

Let's talk about an entirely hypothetical deployment scenario, where you've got a fleet of workstations, some kind of workstation management system that those workstations are plugged in to, an empty AWS account, and an insatiable thirst to secure the enterprise. You can leverage these assets in the following fashion:

1. Go to https://bitbucket.org/asecurityteam/spacecrab and clone the repo to your local machine.

2. Follow the instructions in the repo until you have a new SPACECRAB instance installed in your AWS account.

3. Write a script in the appropriate language for your workstations, that talks to your API gateway with your API token you've just made (Step 2 covers a lot of things), and stores the results somewhere on the workstation's file system, in a place an attacker would look. This might be ~/Downloads/accessKeys.csv, or somewhere on the windows desktop, or really anywhere useful. It's entirely up to you and your expectations around attacker behaviour.

4. Sit back and wait to get paged when one of those tokens is immediately used by an inquisitive bear or panda or something else touching your stuff.

Having gone to all this trouble, you can now also add that script to your cloud service deployment pipeline, ensuring there's a set of extremely juicy looking variables waiting for the next person to get remote code execution on your service. Or add it to all your private repositories with a commit hook, or... you'll find somewhere to put them.

III. How does it work?

SPACECRAB has several components, but the ones we care about are lambdas, data stores, API gateways and policies. The lambdas (which is AWS-speak for 'a python script in the cloud') perform most of the grunt work. They're the part that makes new tokens, updates old tokens, queries the data store for metadata and fires alerts, etc. The data store keeps records of the tokens you've created - things like the token identifier as well as user-controlled metadata like "Location" (i.e. where this token was deployed to) and "Owner" (i.e. whose problem is it if this token is ever used). These fields are accessible via the API gateway, which is just a small https wrapper around

some of the lambdas. The last part is the policies. The policy provided for every token SPACECRAB generates is 'for any resource, for any action, deny it'. This is extremely secure, as you can imagine.

The functional flow of a SPACECRAB alert is: someone finds an AWS token, and tries to use it. They receive an error, and go about their day. In the meantime, that failure has been logged, and is stored in an S3 bucket. The log file is then processed by a lambda function, which goes through looking for AWS keys stored in the database. If it finds any, it adds the metadata for that key to the data from the failed event, and pipes it into an alerting pipeline. The alerting pipeline then triggers any number of additional lambdas, which can do things like email you that something bad happened, or alternatively call you at 4am to tell you someone broke into your CISO's desktop again. If you'd like some other alerting action that isn't as loud, you can have it, as long as you can write an AWS lambda function to do it.

In addition to the token metadata, alerts contain event data, which includes useful fields such as 'source IP' and 'user-agent', as well as the attempted resource and action (i.e. 's3 list-buckets'). With your metadata, you should now have the source of the breach, as well as information about where the breacher came from, and what they were trying to do. They will also, hopefully, not know that they've been rumbled, so you can set up that anthropological study you've always wanted to do and write the next The Cuckoo's Egg (or just roll your incident response plan, whatever works for you).

IV. But does it actually work?

Yes. There's actually an unanticipated side-benefit of deploying honey tokens throughout your network, which is that if people (red teamers, pen testers, attackers generally) find out you're doing it, they get *really nervous*. Maybe even nervous enough to walk away from legitimate credentials, and then the whole operation, saving you a lot of trouble.

But the first and most important benefit, letting you know when something you care about has been breached, definitely works.

To test this, we deployed a lot of AWS credentials to some little-known content hosting solutions, every half an hour for a week or so. All of these tokens are 'breached', of course, because they're on the internet without any authentication in front of them, and you can probably find them in search engines now. But not all of the breached tokens were actually used, at least not in the window that they were valid tokens.

We found some interesting patterns, however:

1. Most attackers will just try one or two actions, and they will usually use a third-party gui client to do this. The vast majority of these attackers are just looking for low hanging fruit like readable/writable S3 buckets, and if they don't find any they will give up immediately.

2. Some attackers are seasoned professionals, and have written scripts to use the AWS CLI tooling, to attempt *every* action an AWS key-pair can take. These attackers aren't just looking for the easy wins; these are the ones that will leverage a tiny information leak into more access and persistence and very expensive server instances if you give them any way in at all.

3. Both of these types of attackers are still dangerous if they actually find anything, though.

In terms of actual use statistics, there was a very clear delineation between the two types of content host. One, an online git repository service, resulted in 82% of 'leaked' tokens being used, and 100% of the used tokens were used 30 minutes (plus or minus 15 seconds) after they were 'leaked'. This speaks to a degree of automation in the process, and (probably) a 30 minute time from repository update to new tokens showing up in the search API.

The second content hosting site, an online paste service, had a wildly different exploitation profile. The 'leaked' credentials were almost certainly recorded within minutes of being published (page views on this site for an otherwise uninteresting paste hit ~60 views in a few minutes, and then are stable over weeks), but no exploitation attempts were made for hours or days (exploitation times were between 12 hours and ~48 hours). Additionally, only 9% of leaked credentials were attempted within the week they were active. This speaks to an ongoing gap in attacker intelligence gathering and potentially a marketable information source for a sufficiently motivated person.

V. Conclusion

This is not really a conclusion, to be honest, this is the premise: honey tokens are an extremely useful way to help secure your network (and your supply chain), and automatable honey tokens are better, because they're easy to deploy.

If you're running an AWS shop, you should consider SPACECRAB for your honey token deployment needs, and if you deploy SPACECRAB you should consider contributing to the project, because we could use the help.