# 2

# EVASION STRATEGIES

To avoid detection, red teams must understand the instrumentation defenders are deploying and where detection teams are likely investing resources. In this chapter, we'll examine the methods detection engineering teams use to discover red team activities and how you can counter them. Remaining undetected requires a multi-layered evasion strategy that includes estimating resources, optimizing timing, blending in, and changing code languages.

## Detection Engineering Patterns

The evolution of enterprise security has spawned specialized teams focused on identifying offensive and anomalous activities. Detection engineering teams often begin as informal Security Operations Center (SOC) groups or infrastructure security staff who spend at least part of their time crafting custom signatures. These teams must also balance real-time detection with long-term analysis. To avoid tying up infrastructure resources during runtime, they typically limit how far back they search for suspicious activity. Their initial detection signatures usually target security information and event management (SIEM) platforms, EDR, and intrusion prevention systems (IPSs), then expand to other tools as the team matures.

Once the team reaches an intermediate level of maturity, it formalizes its work into a program—usually known as *Detection as Code (DaC)*—and pushes detections at a faster pace, typically in one- or two-week sprints. The DaC mechanism includes continuous integration and continuous delivery (CI/CD) pipelines with layers of testing and version control. Due to constraints on compute and testing resources, most of these pipelines limit validation to ensuring that detection rules correctly identify known malicious activities (true positives) and typically *exclude* tests for missed detections (false negatives).

The most advanced detection engineering teams focus on content development upstream of SIEM, such as EDR and other data sources that feed monitoring. To complement EDR, these teams combine OS logs with enriched logging such as Windows Sysmon, creating redundancy in case of a failure. At this maturity stage, indicators are less of a focus as the team prioritizes tactics, techniques, and procedures (TTPs) and activity that deviates from behavioral baselines. Detection engineering at this level often incorporates machine learning approaches, classification models, and regression analysis to better identify anomalies.

How much effort you need to invest in evasion throughout the kill chain, therefore, depends on the detection team's level of maturity. If the team that supports the organization is new or a managed security service provider (MSSP), the bar is lower: You can focus on avoiding typical indicators, such as known string payloads and infrastructure, and you may need to change your implant payload or techniques only once a week. You can estimate the target organization's detection capabilities by reviewing its job postings while preparing your campaign.

## SOC Patterns

Whereas detection engineering teams focus on correlation of static content, the SOC is shift-oriented and operates on a first-in, first-out mentality driven by SLAs and KPIs that reward rapid response and analysis. Many modern SOC environments receive large volumes of *benign* positives, which they consider noise. The clash between detection engineering and SOC teams centers on negotiating alert tuning, usually in the form of events- or alerts-per-hour averages.

Alert fatigue can kick in when alert volumes are high, and these alerts are usually handled by less experienced analysts. Fatigued analysts, assuming that similar indicators represent the same threat, are more likely to close alerts in bulk instead of investigating each one thoroughly. Some SOCs also have permissions to reduce noise by adding or removing atomic-level indicators from pass lists in their tools, such as the SIEM, Network IPS, and EDR.

During shift handovers, SOC analysts brief the next team coming online on any behaviors or anomalies they've noticed in the last 8 to 12 hours. SOC leadership, if centralized, may not piece together trends between shifts for up to a day, or even longer during weekends and holidays. While some SOCs conduct their own threat-hunting activities, many are not deeply trained and rely primarily on basic indicator searches in their SIEM, EDR, and Network IPS logs.

The SOC's operating model is to work as many cases as practical in a finite amount of time and then move on. This limited bandwidth makes it difficult to maintain situational awareness of activities from other teams, such as vulnerability management, risk, cyber threat intelligence, detection engineering, and announced penetration testing.

Understanding these operational limitations gives you a strategic advantage when planning red team activities. Next, we'll explore some ways to leverage this knowledge in your campaign planning.

## Estimating Resources

When developing an initial evasion strategy, you must consider not only the likely capability limits of the detection engineering and SOC teams but also your own resource constraints. Prioritize what you can reasonably accomplish while crafting and executing attacks. The most common pitfall for both defending teams is directing nearly all their efforts to SIEM and EDR. Both tend to ignore enumeration activities and focus instead on well-established LOLBins; malicious execution; and Create, Read, Update, Delete (CRUD) operations that change privilege stages.

Many SIEM detection rules search back no further than one hour, with scans running every 15 to 30 minutes for coverage overlap. To avoid triggering logs that would typically appear in OS user space, stagger your executions outside this one-hour window. This reduces the likelihood of triggering correlation alerts in SIEM and EDR systems and drawing the attention of SOC staff in busy environments.

SIEM and EDR are the primary tools for most SOC and detection engineering teams but typically detect activities toward the end of the kill chain. Many enterprises limit logging volume due to cost. Upstream alerts usually come first—such as from EDR, IPS, and web application firewalls (WAFs)—followed by OS or application-level logging that matches well-known out-of-the-box signatures. For your campaign, prioritize hiding execution activities and system posture changes, which are more likely to be detected. In earlier stages, rotating indicators may suffice instead of hiding patterns.

## Optimizing the Timing of Your Operations

To reduce correlation risk by an analyst or SIEM, you can stretch out the timing of your enumeration and delivery activities. Changing your indicator timing—enumerating during typical business hours and running your exploit during skeleton crew shifts—can thwart analysts and detection teams attempting to correlate your activities.

Simply modifying your campaign's operating timing isn't enough, however. You also need to build that capability into your tooling. If SIEMs operate in 15- to 60-minute increments, EDRs typically operate in seconds to minutes. For example, Microsoft Defender for Endpoint has a default policy of 10 seconds for blocking unknown binaries (*https://learn.microsoft.com/en-us/defender-endpoint/respond-file-alerts*). You're running down the clock by making educated guesses about vendor capabilities, detection engineering optimization, and the target organization's policy tuning.

Many cloud-based EDRs and SIEMs retroactively correlate any additional unknown behavior in sandboxes, searching in a limited one- to three-hour window and then returning a risk assessment based on automated and human analysis. If you suspect your payload will trigger at least one flagged operation, you should assume you have four hours or less to adjust your tooling TTPs and move laterally to avoid risking the rest of the campaign.

Once you've established your ideal operation cadence, you should add tool randomization. Timing-related randomization makes it more difficult for EDR and SIEMs to establish appropriate baselines for traffic and endpoint activities. For example, if the target's detection systems typically correlate at 15-minute intervals with a 60-minute lookback window, you could strategically set your operation intervals between 16 and 77 minutes. This range falls just outside normal detection parameters, complicating correlation. Effective tool randomization requires proper seeding of your randomization function, as shown in Listing 2-1.

```
import time, os, random
# Example seeding for pseudorandom number generator
seed = int(time.time() * 1000) ^ os.getpid()
random.seed(seed)
number = random.randint(16, 77)
print(number)
```

*Listing 2-1: Dynamic seeding for random number generation*

When considering long intervals between events, such as implant beaconing, you must account for asynchronous state tracking and communication windows. Depending on your desired persistence and evasion time, you could batch sequenced payloads at runtime with appropriate windows between events.

## Blending In

While timing strategies help evade correlation-based detection, they're only one part of a comprehensive approach. To further reduce detection risk, you need to blend into the target environment.

### *Using Low-Entropy Encryption*

Regardless of where they occur, detections generally fall into two categories. The first involves typical static indicators—anything searchable as a string. The second is behavior driven, where event sequences increase risk scores beyond baseline operations. Unfortunately, many red team tools fail because they're too focused on masking indicators or executing operations with true entropy (perfect randomness).

Many EDRs and some SIEMs actively scan for high levels of entropy and well-known encoding standards (such as Base64, hexadecimal, and strong cryptography). Preprocessors and normalization modules can easily flag your tooling when it includes encoding and an encrypted payload. To blend in better, consider modifying your static indicators to maintain average Shannon Entropy Scale values between 1 and 4 (the scale ranges from 1 to 8). Doing so will decrease your chance of triggering detection based on a high entropy value.

You should always assume your exploit or payload will eventually burn during the campaign. Most red team activities prioritize short-term evasion over nation-state persistence, so you're primarily concerned with making forensic analysis just difficult enough to survive the current—and possibly next—campaign. But if you shouldn't use Base64, hexadecimal, and strong cryptography, what *should* you use?

We've been successful creating our own weaker encryption system using a rotating symmetric key and a simple substitution cipher. A cryptographically weak cipher that mimics human language levels on the Shannon Entropy Scale bypasses many in-transit and at-rest scanners. For example, in a script with standard ASCII characters, you could map each character to increasing integers, then apply a simple linear function with a key to transform plaintext to ciphertext.

Here's a working example from our code (*https://github.com/dc401/mixed_scripts/blob/master/chowencryption.py*). The snippets use the simple function $3x + k$ for the encryption process:

```
def chowencrypt(cleartext, key):
    # Data dictionary of common text and CLI chars
    encodeddict = {
        'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4,
        'e' : 5, 'f' : 6, 'g' : 7, 'h' : 8,
        'i' : 9, 'j' : 10, 'k' : 11, 'l' : 12,
--snip--
    # Use encryption algo to convert encoded data to cipher text.
    # Weak algo: 3x + key for demo purposes.
    cipherstream = []
```

```
    # Prepend the initialization vector unencrypted first to later combine with the key.
    cipherstream.append(iv)
    # Our new key is the composite addition of IV + key.
    compositekey = iv + int(key)
    for i in encodedbuffer:
        encryptedbyte = (3 * i) + int(compositekey)
        cipherstream.append(encryptedbyte)

    print('encrypted string: ' + str(cipherstream))
--snip--
A low-entropy cipher
```

Building in weak encryption with relatively low entropy still masks indicators, including strings like LOLBin commands, while the payload remains at rest. If you decrypt using an inverse function of $(x - k) / 3$ as shown in Listing 2-2, the payload will never be displayed in cleartext.

```
--snip--
    # We need to read the 'cleartext' IV in the first element of the list.
    # The IV will combine with the user-specified key to decrypt the stream appropriately.
    readiv = encodedbuffer[0]
    compositekey = int(readiv) + int(key)
    for i in encodedbuffer:
        decryptedsignal.append(int((i - int(compositekey)) / 3))
    print('decrypted signal: ' + str(decryptedsignal))

    # Return the decrypted codes to the original ASCII equiv
    decryptedtext = []
    for i in decryptedsignal:
        # Remember encodeddict is a dictionary using key value pairs
        # Must access via .items method for value to key
        for k,v in encodeddict.items():
            if v == i:
                decryptedtext.append(k)
    print('decrypted string as list: ' + str(decryptedtext))
    # Convert the list decryptedtext into original string form
    decryptedtextstring = ''
    for i in decryptedtext:
        decryptedtextstring = decryptedtextstring + str(i)

    print('decrypted string original: ' + str(decryptedtextstring))
    return decryptedtextstring
--snip--
```

*Listing 2-2: A decryption routine for a low-entropy cipher*

With a little time, digital forensics and incident response (DFIR) analysts will eventually run the iterations and discover this pattern, but automated detection remains unlikely at this writing.

Although Listing 2-2 uses ASCII, you could easily adapt it for binary data by converting to hexadecimal first, then encrypting the hex stream with a low-entropy cipher, as shown in Listing 2-3. This requires creating a

key-value database for the hex characters by assigning them new offset integers, just like a standard alphabet (0–9, A–F).

```
# Extending values from 0x00 to 0xff
hex_dict = {hex(i)[2:].zfill(2): i + 500 for i in range(256)}

# New dictionary kv pairs
encoded_dict = {
    # ...existing character mappings
    'a': 1, 'b': 2,  # etc...
    # Hex mappings (will map 00-ff to values starting at 500)
    '00': 500, '01': 501, '02': 502,  # etc...
    'ff': 755
}
--snip--
```

*Listing 2-3: Adding hex support for binary low-entropy encryption*

Now you'll combine everything into a fully working script that will support hex-encoded binary input. In Linux, you can set up any binary into a hex representation using a one-liner set of pipes and existing utilities, as shown in Listing 2-4.

```
xxd -p /path/to/bin | tr -d '\n' | python3 main.py
```

*Listing 2-4: Converting the binary into a hex stream*

After normalizing the hex string by removing newline characters, you pipe it into a script using standard input (stdin). This script, shown in Listing 2-5, will print to standard output (stdout) for demonstration purposes, but you can easily modify it to redirect to a file.

```
#!/usr/bin/env python3
import sys
from random import randint

def hex_encrypt(hex_input, key=10):
    # Convert hex string to integers.
    hex_bytes = [int(hex_input[i:i+2], 16) for i in range(0, len(hex_input), 2)
if len(hex_input[i:i+2]) == 2]

    iv = randint(311, 457)
    composite_key = iv + int(key)

    cipher = [iv]
    for byte in hex_bytes:
        encrypted = (3 * byte) + composite_key
        cipher.append(encrypted)

    return cipher

def hex_decrypt(cipher_list, key=10):
    iv = cipher_list[0]
    composite_key = iv + int(key)
```

```
    decrypted = []
    for value in cipher_list[1:]:
        original = int((value - composite_key) / 3)
        decrypted.append(original)

    return ''.join([format(b, '02x') for b in decrypted])

if __name__ == '__main__':
    # Read from stdin.
    hex_input = sys.stdin.read().strip()

    # Encrypt
    encrypted = hex_encrypt(hex_input)
    print(f"Encrypted: {encrypted}")

    # Decrypt
    decrypted = hex_decrypt(encrypted)
    print(f"Decrypted hex: {decrypted}")
```

*Listing 2-5: Encrypting and decrypting the hex-represented binary*

This script takes your hex input, encrypts it using your low-entropy cipher, and places the result into a list of elements that can later be decrypted and executed as needed.

You can test this script with any payload by piping hex-encoded data to stdin, as shown in Listing 2-6; it will work with any hex-represented string, not just binary data.

```
echo "Hello World" | xxd -p - | tr -d '\n' | python3 main.py
Encrypted: [441, 667, 754, 775, 775, 784, 547, 712, 784, 793, 775, 751, 481]
Decrypted hex: 48656c6c6f20576f726c640a
```

*Listing 2-6: Processing stdin through the hex encryption pipeline in Linux*

Some use cases for low-entropy binaries include evading data loss prevention (DLP) over the wire or executing payloads with files in memory or on disk during EDR scans.

## Avoiding Risky Binaries and Packages

While not every campaign will need fully custom shells, it's generally a good idea to avoid LOLBins, GTFOBins, and certain code libraries that are monitored or hooked for indicator inspection. Some notorious examples—which are especially vulnerable when combined—are Windows utilities like rundll .exe, cmd.exe, and PowerShell; Linux shells like bash; and networking and cryptographic modules. Spawning a known command line shell that has many subprocesses will trigger most EDRs.

To reduce the chances of triggering an alert, consider using spaced-out, properly timed system calls that mimic typical execution patterns, along with the low-entropy encrypted payload just discussed. Listing 2-7 demonstrates a proof of concept to mimic common functions without launching a shell.

```
#!/usr/bin/env python3
import os
import time
import random
import pwd
import grp

def get_user_info():
    # Get current process user info (equivalent of 'id' command)
    uid = os.getuid()
    gid = os.getgid()
    user = pwd.getpwuid(uid).pw_name
    groups = [g.gr_name for g in grp.getgrall() if user in g.gr_mem]
    primary_group = grp.getgrgid(gid).gr_name

    return f"uid={uid}({user}) gid={gid}({primary_group}) groups={','.join(groups)}"


def enum_directory():
    try:
        with os.scandir("/home") as entries:
            for entry in entries:
                print(entry.name)
                # Add small random delay between each entry to avoid pattern
                time.sleep(random.uniform(0.1, 0.3))
    except PermissionError:
        pass


# Run once with delay between operations.
enum_directory()

# Single random delay between 5 and 7 minutes
delay = random.uniform(300, 420)
time.sleep(delay)

# Get user info once.
user_info = get_user_info()
print(user_info)
```

*Listing 2-7: Using Linux-based syscalls for `ls` and `id` equivalents*

This script is equivalent to running the `ls` command on */home* and the `id` command.

When running the script as payload, you will see output in two lines separated by the randomized delay you've specified, as shown in Listing 2-8.

```
runner
uid=1000(runner) gid=1000(runner) groups=
```

*Listing 2-8: Executing commands with randomized delays in repl.it containers to evade EDR detection*

This approach helps avoid basic LOLBin calls tracked by an EDR and reduces the risk of correlated indicators based on timing.

### Following Naming Conventions

While giving files and payload elements benign names is not specific to tool development, it reduces risk in several ways. Depending on policy, some security tools exclude certain file extensions from scanning. Unless you're relying on the victim to interactively run your payload, you should use extensions that are less likely to be scanned or handled like binaries, such as *.tmp*.

Another approach is mimicking legitimate system files in your naming. On Windows, for example, naming a file *lsaio.exe* helps it blend in since it resembles a legitimate security-related binary—one that isolates the Windows local security authority process in user mode—that isn't always enabled by default.

### Padding Files

In the past, researchers and testers made payloads as small as possible due to memory constraints in early exploitation techniques. The days of MS08-067 and MS17-010 are mostly gone. Modern endpoints and network devices have significantly more operating memory. In fact, padding your payload to 10MB to 20MB isn't a bad idea, since some EDR policies skip scanning files over certain size thresholds—making this effective for loading payloads into memory or writing to disk.

If an executable payload is under 1MB, we recommend padding. Static arrays in compiled languages effectively add legitimate but unused padding to a binary's `.data` section, as shown in Listing 2-9.

```
#include <stdio.h>

// Create array for 1MB.
char padding[1024 * 1024] = {[0 ...(1024 * 1024 - 1)] = 1};

int main() {
    // Use the array so compiler does not optimize.
    printf("Padding starts with: %d\n", padding[0]);
    return 0;
}
~/workspace$ gcc main.c -o main.o
~/workspace$ ls -lah main.o
-rwxr-xr-x 1 runner runner 1.1M Jun 18 03:43 main.o
~/workspace$
```

*Listing 2-9: Adding 1MB of data at compile time in C*

This file-sizing strategy works when you need to exceed the practical inspection limits during network transport or disk writing. Base the padding size on the policies and capabilities of your target that you identified during your campaign's reconnaissance phase.

### *Adding Code Signing*

Code signing, when implemented carefully, is also an effective way to blend in and reduce your detection risk. Modern EDRs and some IPS engines compare certificate information to filenames to look for mismatches. Some red teams use stolen or self-signed certificates, but we advise against this—modern detection tools analyze certificate metadata for anomalies in dates, filenames, and hashes.

Code-signing certificate from trusted vendors such as DigiCert, Verisign, or Thawte start at around $500 USD and require USB hardware or cloud-backed hardware security modules (HSMs). Signing tools differ by OS but need appropriate development environment libraries. Listing 2-10 uses signtool.exe with a DigiCert certificate to sign a Windows binary.

```
signtool.exe sign /csp "DigiCert Signing Manager KSP" /kc key1 /f example.crt /tr
http://timestamp.digicert.com /td SHA256 /fd SHA256 signthis.util.exe
```

*Listing 2-10: Using DigiCert to sign Windows binaries*

The signing process accesses a time server to validate the certificate. Avoid signing on target hosts during campaigns. Detection systems may consider certificate age domain registration date, so letting certificates "bake" for months to a year benefits well-planned campaigns.

## Switching Languages and Architecture

Developing evasive payloads will always be a cat-and-mouse game. The more documented resources there are, the greater the chance that tools and content will detect your activities. You can mitigate this risk by switching to a less popular coding language and compiler. For example, mature reverse engineering and forensic tools have been developed for both C and Delphi, so tooling in those language families isn't recommended. Go, however, is a newer language for which defenders have had less time to create analysis tools, and it can operate in interpreted or compiled mode. While tools like Ghidra (with extensions) and Mandiant's GoReSym do support Go analysis, these capabilities are typically less mature than those for analyzing other languages. Some of the main differences that make Go more unique to analyze is its use of split-stack management, its self-contained runtime when compiled, and the way it handles string structures.

From a forensics perspective, Go contains rich metadata that can aid analysis, which is why some long-term nation-state malware still uses C-family languages to reduce reverse engineering artifacts. Every campaign is different, however. Stripped binaries (those with all symbols removed) can look even more suspicious to sandboxes and EDRs during static property analysis.

Figure 2-1 compares binary structures in C and Go.

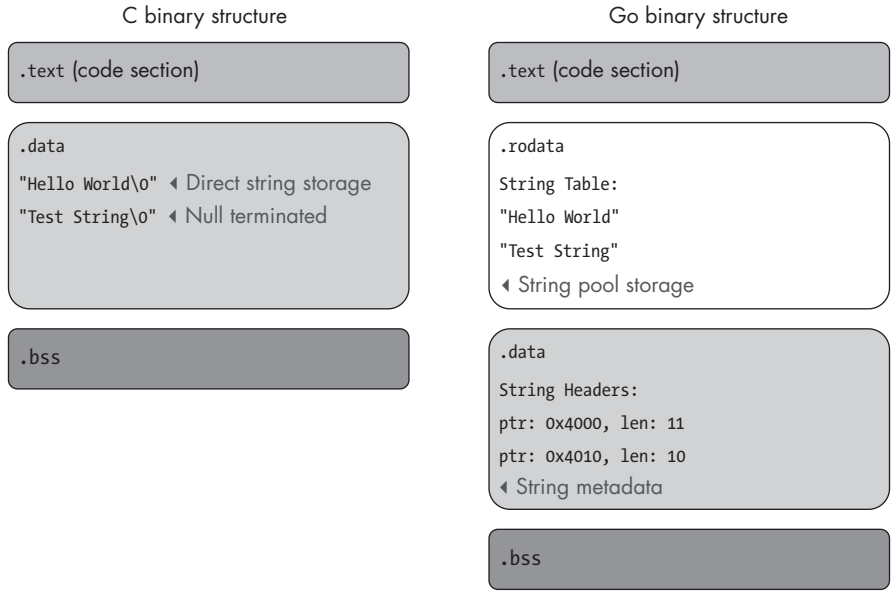


*Figure 2-1: Comparing C and Go binary string storage methods*

The C binary stores strings with a null-terminated byte, whereas the Go binary contains metadata that references string pointers. Although Go provides additional metadata, tools without native support will be limited in their detection capabilities when parsing these binaries. Beyond static property analysis, static function and control flow analysis also differ greatly between the two languages.

As EDRs advance with the help of their cloud-enabled correlations, it's important to also consider the target's likely architecture. Most environments we've encountered use Apple Silica ARM-64, IoT general ARM-64, and Intel x86-64. We've also found that 64-bit compiled binaries tend to trigger fewer alerts than their 32-bit counterparts. This may be because EDRs assign a higher risk score to 32-bit binaries, based on malware authors in the past using 32-bit for widespread compatibility when delivering payloads at scale.

## Other Evasion Techniques

Generally, the more layers of evasion you apply between development and runtime, the higher the complexity barrier for analysts trying to triage forensic indicators. When selecting which techniques to pursue, determine the ratio of likely mean time to detection to your total development time. You don't want to spend hours developing an evasion technique that only delays detection by minutes.

Adding layers of evasion is the same strategy blue teams use in defense. While not every technique mentioned in this chapter is required, we recommend, at a minimum, steering clear of LOLBins, using non–C-based coding languages, and randomizing timing delays. Consider every target organization's unique skill level, which you can typically glean from its security team's public posts or contributions.

### Mimicking End User Behavior

During payload and tool development, mimicking the target environment might include replicating how end users or IT teams utilize the target assets. Time permitting, add subroutines that a legitimate long-term application performs beyond typical C2 implant traffic. For example, introduce web surfing traffic initially to the endpoint's top DNS resolver cache domains before initiating beacon activity. During scanning or runtime, basic API operations to read system date and time, and calls to benign functions such as basic calculations or string manipulations, can also affect EDR risk scores.

### Distracting Defenders with Tandem Operations

Although not a tool-specific technique, consider taking advantage of volume-based noise and SOC alert fatigue as part of your campaign execution strategy. Deploy payloads and indicators you're comfortable "burning" to use up defenders' valuable time and mental energy. Many SOCs become highly focused on a single set of indicators when you launch attacks and exploits that mimic what's in the news. In the meantime, your main campaign can continue using completely different TTPs without garnering much attention.

### Using False Flags for Misdirection

Some red teams use false flags with tandem operations to misdirect threat intelligence by attributing unique indicators and TTPs to a known threat group. As part of a layered evasion strategy, false flags can prolong analysis, giving your team more time to wrap up operations. It's best to inject false flags in burnable payloads, since you know it will be actively analyzed by mature defenders.

Less experienced defenders might dismiss false flags in string keywords that resemble known security vendors or tools, attributing them to routine security operations such as vulnerability scanning and penetration testing. An alert-fatigued analyst who examines a single indicator that says Qualys or Rapid7, for example, might ignore traffic going to unrelated addresses.

## Summary

In this chapter, we covered multiple techniques for building an evasion strategy during tool development and runtime. Understanding typical defender resource constraints helps you prioritize which evasions warrant more development time. You learned how practical timing randomization and delays can reduce the chances of a defender correlating successful routines, as well as how lower-entropy obfuscation using custom encryption can evade anomaly-based detections. Finally, we discussed the benefits of layering multiple techniques—including following naming conventions, sizing files appropriately, adding code signing, and strategically using intentionally detectable TTPs to redirect the defending team's focus—as part of a comprehensive evasion strategy.