Özelleştirme

- Auxiliary Modülleri
- Machinery Modülleri
- Analiz Paketleri
- Processing Modülleri
- İmzalar
- Raporlama Modülleri

Auxiliary Modülleri

Yardımcı modüller, her tek analiz sürecine paralel olarak yürütülmesi gereken bazı prosedürleri tanımlar. Tüm yardımcı modüllerin `cuckoo/cuckoo/auxiliary/` dizini altına yerleştirilmesi gerekir, bu şekilde modül `cuckoo.auxiliary` modülü altına düşer.

Bir modülün yapısı aşağıdakine benzerdir:

```
from cuckoo.common.abstracts import Auxiliary

class MyAuxiliary(Auxiliary):

    def start(self):
        # Do something.

    def stop(self):
        # Stop the execution.
```

`start()` fonksiyonu, analiz makinesini başlatmadan önce ve gönderilen kötü amaçlı dosyayı etkili bir şekilde yürütmek üzere başlatılacaktır, `stop()` fonksiyonu analiz sürecinin sonunda, işleme ve raporlama prosedürlerini başlatmadan önce başlatılacaktır.

Örneğin, Cuckoo tarafından varsayılan olarak sağlanan bir yardımcı modül `sniffer.py` olarak adlandırılır ve oluşturulan ağ trafiğini dump etmek için tcpdump'ı çalıştırmaktan sorumludur.

Machinery Modülleri

Machinery modülleri, Cuckoo'nun sanallaştırma yazılımınızla nasıl etkileşimde bulunması gerektiğini tanımlar (veya potansiyel olarak fiziksel disk görüntüleme çözümleriyle bile). Belirli bir vendor zorlamama kararı aldığımızdan beri, sürüm 0.4'ten itibaren tercih ettiğiniz çözümü kullanabilir ve varsayılan olarak desteklenmiyorsa, Cuckoo'nun nasıl kullanılacağını tanımlayan özel bir Python modülü yazabilirsiniz.

Her makine modülü, `cuckoo/cuckoo/machinery/` dizini içinde bulunmalıdır, böylece `cuckoo.machinery` modülü altına düşer.

Temel bir machinary modülü şöyle görünecektir:

```
from cuckoo.common.abstracts import Machinery
from cuckoo.common.exceptions import CuckooMachineError

class MyMachinery(Machinery):
    def start(self, label):
        try:
            revert(label)
            start(label)
        except SomethingBadHappens:
            raise CuckooMachineError("oops!")

    def stop(self, label):
        try:
            stop(label)
        except SomethingBadHappens:
            raise CuckooMachineError("oops!")
```

Cuckoo için gereksinimler şunlardır:

- Class, `Machinery`'den alınmalıdır.
- `start()` ve `stop()` fonksiyonlarınız olmalıdır.
- Bir şey başarısız olduğunda `CuckooMachineError`'u yükseltmelisiniz.

Anlaşıldığı gibi, makine modülü bir Cuckoo kurulumunun temel bir parçasıdır, bu nedenle kodunuzu hata ayıklamak için yeterli zaman harcadığınızdan ve onu herhangi beklenmedik bir hataya karşı sağlam ve dirençli hale getirdiğinizden emin olun.

Konfigürasyon

Her makine modülü, `$CWD/conf/<makine modülü adı>.conf` (bu, Git deposundaki `cuckoo/data/conf/<makine>conf`'ye çevrilir) konumunda özel bir yapılandırma dosyasıyla birlikte gelmelidir. Örneğin, `cuckoo/cuckoo/machinery/kvm.py` için bir `$CWD/conf/kvm.conf`'ya sahibiz.

Yapılandırma dosyası varsayılan yapıyı izlemelidir:

```
[kvm]
# Specify a comma-separated list of available machines to be used. For each
# specified ID you have to define a dedicated section containing the details
# on the respective machine. (E.g. cuckoo1,cuckoo2,cuckoo3)
machines = cuckoo1

[cuckoo1]
# Specify the label name of the current machine as specified in your
# libvirt configuration.
label = cuckoo1

# Specify the operating system platform used by current machine
# [windows/darwin/linux].
platform = windows

# Specify the IP address of the current machine. Make sure that the IP address
# is valid and that the host machine is able to reach it. If not, the analysis
# will fail.
ip = 192.168.122.105
```

Bu konfigürasyon dosyasında, bir [<modül adı>] başlığı bulunmalı ve bir makine ID'lerini içeren bir `'machines'` alanına sahip olmalıdır.

Her bir makine için bir etiket, bir platform ve IP belirtmelisiniz.

Bu alanlar, Cuckoo'nun zaten gömülü `initialize()` fonksiyonunu kullanması için gereklidir. Bu fonksiyon, mevcut makinelerin listesini oluşturur.

Konfigürasyon yapısını değiştirmeyi planlıyorsanız, `initialize()` fonksiyonunu geçersiz kılmalısınız (kendi modülünüz içinde, Cuckoo'nun çekirdek kodunu değiştirmeniz gerekmez). Orijinal kodunu `cuckoo/common/abstracts.py` içindeki `Machinery` classta bulabilirsiniz.

LibVirt

Cuckoo 0.5 sürümüyle başlayarak, LibVirt üzerine yeni makine modülleri geliştirmek oldukça kolaydır. `cuckoo/common/abstracts.py` dosyasında zaten gerekli tüm işlevselliği sağlayan `LibVirtMachinery`'i bulabilirsiniz. Bu temel sınıftan miras alarak, aşağıdaki örnekte olduğu gibi bağlantı dizesini belirtmeniz yeterlidir:

Cuckoo 2.0.7a1 sürümüyle başlayarak, özel bir dsn kullanabilirsiniz. Bu dsn'yi kvm.conf dosyasında ayarlayarak kullanabilirsiniz. Örnek:

dsn = qemu+ssh://192.168.56.1/system

Bu, LibVirt tarafından desteklenen tüm sanallaştırma teknolojileri için çalışır. Sadece, LibVirt paketinizin (örneğin, Linux dağıtımınızdan kullanıyorsanız) ihtiyacınız olan teknolojiyi destekleyecek şekilde derlendiğinden emin olun.

Aşağıdaki komutla kontrol edebilirsiniz:

```
$ virsh -V
Virsh command line tool of libvirt 0.9.13
See web site at http://libvirt.org/

Compiled with support for:
 Hypervisors: QEmu/KVM LXC UML Xen OpenVZ VMWare Test
 Networking: Remote Daemon Network Bridging Interface Nwfilter VirtualPort
 Storage: Dir Disk Filesystem SCSI Multipath iSCSI LVM
 Miscellaneous: Nodedev AppArmor Secrets Debug Readline Modular
```

Eğer sanallaştırma teknolojiniz `Hypervisors` listesinde bulunmuyorsa, eksik olanı belirli destekle tekrar derlemeniz gerekecektir.

Analiz Paketleri

Analiz Paketleri bölümünde açıklandığı gibi, analiz paketleri, Cuckoo'nun analizör bileşeninin bir konuk ortamındaki belirli bir dosya için analiz prosedürünü nasıl yürütmesi gerektiğini açıklayan yapılandırılmış Python sınıflarıdır.

Zaten bildiğiniz gibi, kendi paketlerinizi oluşturabilir ve bunları varsayılan olanlarla birlikte ekleyebilirsiniz. Yeni paketler tasarlamak çok kolaydır ve sadece temel düzeyde programlama ve Python dilini anlama gerektirir.

Başlarken

Bir örnek olarak, genel Windows uygulamalarını analiz etmek için varsayılan paketi inceleyeceğiz. Bu paket, `$CWD/analyzer/windows/packages/exe.py` konumunda bulunur (Git deposunda `cuckoo/data/analyzer/windows/packages/exe.py` olarak çevrilir):

```
from lib.common.abstracts import Package

class Exe(Package):
    """EXE analysis package."""

    def start(self, path):
        args = self.options.get("arguments")
        return self.execute(path, args)
```

Bu gerçekten kolay görünüyor, Package nesnesinden alınan tüm yöntemler sayesinde. Bir analiz paketinin Package nesnesinden aldığı başlıca yöntemlere bir göz atalım:

```
from lib.api.process import Process
from lib.common.exceptions import CuckooPackageError

class Package(object):
    def start(self):
        raise NotImplementedError
```

```
        def check(self):
            return True

        def execute(self, path, args):
            dll = self.options.get("dll")
            free = self.options.get("free")
            suspended = True
            if free:
                suspended = False

            p = Process()
            if not p.execute(path=path, args=args, suspended=suspended):
                raise CuckooPackageError(
                    "Unable to execute the initial process, analysis aborted."
                )

            if not free and suspended:
                p.inject(dll)
                p.resume()
                p.close()
                return p.pid

        def finish(self):
            if self.options.get("procmemdump"):
                for pid in self.pids:
                    p = Process(pid=pid)
                    p.dump_memory()
            return True
```

Kodu inceleyelim:

- Satır 1: Windows işlemlerini oluşturup manipüle etmek için kullanılan Process API sınıfını içe aktarın.
- Satır 2: Paketin yürütülmesi sırasında analizciye sorunları bildirmek için kullanılan CuckooPackageError istisnasını içe aktarın.
- Satır 4: Ana sınıfı tanımlayın, object sınıfından miras alacak şekilde.
- Satır 5: path adlı dosyanın yürütülmesini sağlayan start() fonksiyonunu tanımlayın. Her analiz paketi tarafından uygulanmalıdır.
- Satır 8: check() fonksiyonunu tanımlayın.
- Satır 13: Sürecin izlenip izlenmeyeceğini tanımlayan free seçeneğini edinin.
- Satır 18: Bir Process örneğini başlatın.
- Satır 19: Zararlı yazılımı çalıştırmaya çalışın, başarısız olursa yürütmeyi durdurun ve analizciye bildirin.
- Satır 24: Sürecin izlenip izlenmemesi kontrol edilir.
- Satır 25: Süreci DLL ile enjekte edin.

- Satır 26: Süreci askıya alınmış durumdan devam ettirin.
- Satır 28: Yeni oluşturulan sürecin PID'sini analizciye döndürün.
- Satır 30: finish() fonksiyonunu tanımlayın.
- Satır 31: procmemdump seçeneğinin etkin olup olmadığını kontrol edin.
- Satır 32: Şu anda izlenen süreçler arasında döngü yapın.
- Satır 33: Bir Process örneğini açın.
- Satır 34: Sürecin belleğinden bir döküm alın."

start()

Bu fonksiyonda çalıştırmak istediğiniz tüm başlatma işlemlerini yerleştirmelisiniz. Bu, zararlı yazılım sürecini çalıştırmayı, ek uygulamaları başlatmayı, bellek anılarını almayı ve daha fazlasını içerebilir.

check()

Bu fonksiyon, zararlı yazılım çalışırken Cuckoo tarafından her saniye çalıştırılır. Bu fonksiyonu, her türlü tekrarlayan işlemi gerçekleştirmek için kullanabilirsiniz.

Örneğin, analizinizde yalnızca belirli bir gösterge oluşturulmasını bekliyorsanız (örneğin, bir dosya), koşulu bu fonksiyona yerleştirebilir ve False döndürürse analiz hemen sona erer.

Bu, "analiz devam etmeli mi yoksa durmalı mı?" olarak düşünülebilir.

Örneğin:

```
def check(self):
    if os.path.exists("C:\\config.bin"):
        return False
    else:
        return True
```

Bu `check()` fonksiyonu, `C:\\\\config.bin` oluşturulduğunda Cuckoo'nun analizi derhal sonlandırmasına neden olacaktır.

execute()

Kötü amaçlı yazılım yürütmeyi tamamlar ve DLL enjeksiyonu ile ilgilenir.

finish()

Bu fonksiyon, analizi sonlandırmadan ve makineyi kapatmadan önce Cuckoo tarafından basitçe çağrılır. Varsayılan olarak, bu fonksiyon tüm izlenen süreçlerin belleğini dökmek için isteğe bağlı bir özelliği içerir.

Ayarlar

Her paket, otomatik olarak tüm kullanıcı tarafından belirtilen seçenekleri içeren bir sözlüğe erişime sahiptir (bkz. Analiz).

Bu tür seçenekler, `self.options` özniteliğinde kullanılabilir hale getirilir. Örneğin, kullanıcının gönderim sırasında aşağıdaki dizesini belirttiğini varsayalım:

```
foo=1,bar=2
```

Seçilen analiz paketi, bu değerlere erişim sağlayacaktır:

```
from lib.common.abstracts import Package

class Example(Package):

    def start(self, path):
        foo = self.options["foo"]
        bar = self.options["bar"]

    def check():
        return True

    def finish():
        return True
```

Bu seçenekler, paketinizin içinde yapılandırmanız gerekebilecek her şey için kullanılabilir.

Process API

Process sınıfı, çeşitli işlemle ilgili özelliklere ve işlevlere erişim sağlar. Analiz paketlerinizde şu şekilde içe aktarabilirsiniz:

```
from lib.api.process import Process
```

Daha sonra bir örneği şu şekilde başlatırsınız:

```
p = Process()
```

Bir yeni süreç oluşturmak yerine mevcut bir süreci açmak istiyorsanız, birden fazla argüman belirleyebilirsiniz:

- pid: Çalışmak istediğiniz sürecin PID'si.
- h_process: Çalışmak istediğiniz sürecin işlem tanıtıcısı.
- thread_id: Çalışmak istediğiniz sürecin thread ID'si.
- h_thread: Çalışmak istediğiniz sürecin thread işlem tanıtıcısı.

Metotlar

Process.open()

Bir çalışan süreç için bir işlem tanıtıcısı açar. İşlemin başarılı olup olmadığı durumunda True veya False değerini döndürür.

Return type:	boolean

Örnek kullanım:

```
p = Process(pid=1234)
p.open()
handle = p.h_process
```

Process.exit_code()

Açılan sürecin çıkış kodunu döndürür. Daha önce yapılmadıysa, exit_code() işlem tanıtıcısını edinmek için open() çağrısını gerçekleştirir.

Return type:	ulong

Örnek kullanım:

```
p = Process(pid=1234)
code = p.exit_code()
```

Process.is_alive()

exit_code() fonksiyonunu çağırır ve dönen kodun STILL_ACTIVE olduğunu kontrol eder, bu da verilen sürecin hala çalıştığı anlamına gelir. True veya False değerini döndürür.

Return type:	boolean

Örnek kullanım:

```
```

```
p = Process(pid=1234)
if p.is_alive():
    print("Still running!")
```

Process.get_parent_pid()

Açılan sürecin üst sürecinin PID'sini döndürür. Daha önce yapılmadıysa, get_parent_pid() işlem tanıtıcısını edinmek için open() çağrısını gerçekleştirir.

Return type:	int

Örnek kullanım:

```
p = Process(pid=1234)
ppid = p.get_parent_pid()
```

Process.execute(*path*[, *args=None*[, *suspended=False*]])

Belirtilen yol üzerindeki dosyayı çalıştırır. İşlem başarılı olursa True, başarısız olursa False değerini döndürür.

Return type:	boolean

Örnek kullanım:

```
p = Process()
p.execute(path="C:\\WINDOWS\\system32\\calc.exe", args="Something",
suspended=True)
```

Process.resume()

Açılan süreci askıya alınmış durumdan devam ettirir. İşlem başarılı olursa True, başarısız olursa False değerini döndürür.

Return type:	boolean

Örnek kullanım:

```
```

```
p = Process()
p.execute(path="C:\\WINDOWS\\system32\\calc.exe", args="Something",
suspended=True)
p.resume()
```

Process.terminate()

Açılan süreci sonlandırır. İşlem başarılı olursa True, başarısız olursa False değerini döndürür.

Return type:	boolean

Örnek kullanım:

```
p = Process(pid=1234)
if p.terminate():
    print("Process terminated!")
else:
    print("Could not terminate the process!")
```

Process.inject([*dll*[, *apc=False*]])

Açılan sürece DLL'imizi enjekte eder. İşlem başarılı olursa True, başarısız olursa False değerini döndürür.

Return type:	boolean

Örnek kullanım:

```
p = Process()
p.execute(path="C:\\WINDOWS\\system32\\calc.exe", args="Something",
suspended=True)
p.inject()
p.resume()
```

Process.dump_memory()

Verilen sürecin bellek alanının bir anlık görüntüsünü alır. İşlem başarılı olursa True, başarısız olursa False değerini döndürür.

Return type:	boolean

Örnek kullanım:

```
p = Process(pid=1234)
p.dump_memory()
```

ads

Processing Modülleri

Cuckoo'nun işleme modülleri, sandbox'ın ürettiği raw sonuçları özelleştirmenize ve daha sonra imza ve raporlama modülleri tarafından kullanılacak genel bir konteynere bilgi eklemenize olanak tanıyan Python betikleridir.

Birçok modül oluşturabilirsiniz, ancak bu modüllerin önceden belirlenmiş bir yapıyı takip etmeleri gerekmektedir ki bunu bu bölümde sunacağız.

Global Konteyner

Bir analiz tamamlandığında, Cuckoo, `cuckoo/processing/` dizininde bulunan tüm işleme modüllerini çağıracaktır; bunların tümü cuckoo.processing modülü altındadır. Oluşturmayı seçtiğiniz herhangi bir ek modülü, bu dizin içine yerleştirmeniz gerekmektedir.

Her modülün ayrıca `$CWD/conf/processing.conf` dosyasında özel bir bölümü olmalıdır: örneğin, cuckoo/processing/foobar.py adında bir modül oluşturursanız, `$CWD/conf/processing.conf` dosyasına aşağıdaki bölümü eklemeniz gerekecektir:

```
[foobar]
enabled = yes
```

Ardından her modül başlatılacak ve çalıştırılacak, dönen veri global bir konteyner olarak adlandıracağımız bir veri yapısına eklenir.

Bu konteyner, sadece tüm modüller tarafından üretilen sonuçları kimlik anahtarlarına göre sınıflandırılmış bir Python sözlüğüdür.

Cuckoo zaten bir dizi varsayılan modül sağlar ve bu modüller, standart bir global konteyner oluşturacaktır. Bu varsayılan modüllerin değiştirilmemesi önemlidir, aksi takdirde oluşan global konteyner yapısı değişir ve raporlama modülleri bu yapısını tanıyamaz ve nihai raporları oluşturmak için kullanılan bilgileri çıkaramaz.

Mevcut varsayılan işleme modülleri şunlardır:

- AnalysisInfo (`cuckoo/processing/analysisinfo.py`) - mevcut analiz hakkında bazı temel bilgiler oluşturur, zaman damgaları, Cuckoo sürümü vb.
- ApkInfo (`cuckoo/processing/apkinfo.py`) - mevcut APK analizi (Android analizi) hakkında bazı temel bilgiler oluşturur.
- Baseline (`cuckoo/processing/baseline.py`) - toplanan bilgilerden temel çizgileri oluşturur.

- BehaviorAnalysis (`cuckoo/processing/behavior.py`) - ham davranış günlüklerini ayrıştırır ve bazı ilk dönüşümler ve yorumlamalar yapar, bunlar arasında tam süreç izleme, davranış özeti ve süreç ağacı bulunur.
- Buffer (`cuckoo/processing/buffer.py`) - bırakılan tampon analizi.
- Debug (`cuckoo/processing/debug.py`) - analizci tarafından oluşturulan hataları ve analysis.log dosyasını içerir.
- Droidmon (`cuckoo/processing/droidmon.py`) - Droidmon günlüklerinden Dinamik API çağrıları Bilgisi çıkarır.
- Dropped (`cuckoo/processing/dropped.py`) - zararlı tarafından bırakılan ve Cuckoo tarafından dökülen dosyalar hakkında bilgi içerir.
- DumpTls (`cuckoo/processing/dumptls.py`) - monitörden çıkarılan TLS anahtar sırlarını ve PCAP'dan çıkarılan anahtar bilgilerini çapraz referanslamak için bir anahtar sırları dosyasını dökmek üzere.
- GooglePlay (`cuckoo/processing/googleplay.py`) - analiz oturumu hakkında Google Play bilgisi.
- Irma (`cuckoo/processing/irma.py`) - IRMA bağlayıcısı.
- Memory (`cuckoo/processing/memory.py`) - tam bir bellek dökümünde Volatility'yi yürütür.
- Misp (`cuckoo/processing/misp.py`) - MISP bağlayıcısı.
- NetworkAnalysis (`cuckoo/processing/network.py`) - PCAP dosyasını ayrıştırır ve DNS trafiği, alanlar, IP'ler, HTTP istekleri, IRC ve SMTP trafiği gibi bazı ağ bilgilerini çıkarır.
- ProcMemory (`cuckoo/processing/procmemory.py`) - süreç bellek dökümü analizi yapar. Not: modül, data/yara/memory/index_memory.yar'daki kullanıcı tanımlı Yara kurallarını işlemek için yeteneklidir. Yara kurallarınızı eklemek için bu dosyayı düzenleyin.
- ProcMon (`cuckoo/processing/procmon.py`) - procmon.exe çıkışından olayları çıkarır.
- Screenshots (`cuckoo/processing/screenshots.py`) - ekran görüntüsü ve OCR analizi.
- Snort (`cuckoo/processing/snort.py`) - Snort işleme modülü.
- StaticAnalysis (`cuckoo/processing/static.py`) - PE32 dosyalarının bazı statik analizlerini gerçekleştirir.
- Strings (`cuckoo/processing/strings.py`) - analiz edilen ikili dosyadan dizgileri çıkarır.
- Suricata (`cuckoo/processing/suricata.py`) - Suricata işleme modülü.
- TargetInfo (`cuckoo/processing/targetinfo.py`) - analiz edilen dosya hakkında bilgi içerir, örneğin karma değerleri.
- VirusTotal (`cuckoo/processing/virustotal.py`) - analiz edilen dosyanın antivirüs imzaları için VirusTotal.com'da arama yapar. Not: Dosya VirusTotal.com'a yüklenmez, dosya daha önce web sitesine yüklenmediyse sonuçlar alınamaz.

Başlarken

Onları Cuckoo'ya kullanılabilir hale getirmek için tüm işleme modülleri `cuckoo/processing/` dizinine yerleştirilmelidir.

Temel bir işleme modülü şöyle görünebilir:

```
from cuckoo.common.abstracts import Processing
```

```
class MyModule(Processing):

    def run(self):
        self.key = "key"
        data = do_something()
        return data
```

Her işleme modülü şunları içermelidir:

- `Processing` sınıfından türetilmiş bir sınıf.
- Bir `run()` fonksiyonu.
- Dönen veriler için alt konteyner olarak kullanılacak adı tanımlayan `self.key` özniteliği.
- Global konteynere eklenen bir veri seti (liste, sözlük, dize vb.).

Ayrıca bir sıra değeri belirleyebilirsiniz, bu size kullanılabilir işleme modüllerini sıralı bir dizide çalıştırma olanağı sağlar. Varsayılan olarak, tüm modüllerin sıra değeri 1 olarak ayarlanmıştır ve alfabetik sırayla çalıştırılır.

Bu değeri değiştirmek istiyorsanız, modülünüz şöyle görünecektir:

```
from cuckoo.common.abstracts import Processing

class MyModule(Processing):
    order = 2

    def run(self):
        self.key = "key"
        data = do_something()
        return data
```

Ayrıca bir işleme modülünü manuel olarak devre dışı bırakabilirsiniz, bunun için `enabled` özniteliğini `False` olarak ayarlamanız gerekmektedir:

```
from cuckoo.common.abstracts import Processing

class MyModule(Processing):
    enabled = False

    def run(self):
        self.key = "key"
        data = do_something()
```

```
        return data
```

İşleme modülleri, verilen analiz için ham sonuçlara erişmek için kullanılabilecek bazı özniteliklerle sağlanır:

- `self.analysis_path`: sonuçları içeren klasörün yolunu belirtir (örneğin, `$CWD/storage/analysis/1`)
- `self.log_path`: `analysis.log` dosyasının yolunu belirtir.
- `self.file_path`: analiz edilen dosyanın yolunu belirtir.
- `self.dropped_path`: bırakılan dosyaları içeren klasörün yolunu belirtir.
- `self.logs_path`: raw davranış günlüklerini içeren klasörün yolunu belirtir.
- `self.shots_path`: ekran görüntülerini içeren klasörün yolunu belirtir.
- `self.pcap_path`: ağ pcap dökümünün yolunu belirtir.
- `self.memory_path`: eğer oluşturulmuşsa tam bellek dökümünün yolunu belirtir.
- `self.pmemory_path`: eğer oluşturulmuşsa süreç belleği dökümlerinin yolunu belirtir.

Bu özniteliklerle Cuckoo tarafından depolanan tüm ham sonuçlara kolayca erişebilmeli ve bunlar üzerinde analitik işlemlerinizi gerçekleştirebilmelisiniz.

Son bir not olarak, modülün bir sorunla karşılaştığında Cuckoo'ya bildirmek istediğiniz bir durumda CuckooProcessingError istisnasını kullanmak iyi bir uygulamadır. Bu, şu şekilde sınıfı içe aktararak yapılabilir:

```
from cuckoo.common.exceptions import CuckooProcessingError
from cuckoo.common.abstracts import Processing

class MyModule(Processing):

    def run(self):
        self.key = "key"

        try:
            data = do_something()
        except SomethingFailed:
            raise CuckooProcessingError("Failed")

        return data
```

İmzalar

Cuckoo ile analiz sonuçlarına karşı çalıştırabileceğiniz özel imzalar oluşturabilir ve bu imzaları kullanarak belirli bir zararlı davranışı veya ilgilendiğiniz bir göstergeyi tanımlayabilirsiniz.

Bu imzalar, analizlere bir bağlam sağlamak için çok kullanışlıdır: sonuçların yorumunu basitleştirmeleri yanı sıra ilgi çekici kötü amaçlı yazılım örneklerini otomatik olarak tanımlamak açısından da önemlidir.

Cuckoo'nun imzalarını kullanabileceğiniz bazı örnekler:

- Bazı benzersiz davranışları (dosya adları veya mutex'lar gibi) izleyerek ilgilendiğiniz belirli bir kötü amaçlı yazılım ailesini tanımlama.
- Kötü amaçlı yazılımın sistemi üzerinde gerçekleştirdiği ilginç değişiklikleri tespit etme, örneğin cihaz sürücüleri kurulumu.
- Bankacılık Truva atları veya fidye yazılımı gibi belirli kötü amaçlı yazılım kategorilerini izole ederek tanımlama, bunlar tarafından genellikle gerçekleştirilen tipik eylemleri izole ederek.
- Örnekleri kötü amaçlı yazılım/bilinmeyen kategorilere sınıflandırma (temiz örnekleri tanımlamak mümkün değildir).

Kendi oluşturduğumuz ve diğer Cuckoo kullanıcıları tarafından oluşturulmuş imzaları Community reposunda bulabilirsiniz.

Başlarken

İmza oluşturma süreci oldukça basittir ve sadece Python programlamaya iyi bir anlayış gerektirir.

Öncelikle, tüm imzaların Cuckoo'nun `cuckoo/cuckoo/signatures/` dizininde veya Community reposunun `modules/signatures/` dizininde (Community reposu hala eski dizin yapısını kullanıyor) bulunması gerekmektedir.

Aşağıdaki temel bir örnek imzadır:

```
from cuckoo.common.abstracts import Signature

class CreatesExe(Signature):
    name = "creates_exe"
    description = "Creates a Windows executable on the filesystem"
    severity = 2
```

```
    categories = ["generic"]
    authors = ["Cuckoo Developers"]
    minimum = "2.0"

    def on_complete(self):
        return self.check_file(pattern=".*\\.exe$", regex=True)
```

Gördüğünüz gibi, yapı gerçekten basit ve diğer modüllerle tutarlıdır. Daha sonra detaylara gireceğiz, ancak Cuckoo'nun 1.2 sürümünden itibaren imza oluşturma sürecini çok daha kolay hale getiren bazı yardımcı işlevler sağlamaktadır.

Bu örnekte, özet içinde erişilen tüm dosyaları kontrol ediyoruz ve 'exe' ile biten bir şey olup olmadığını kontrol ediyoruz: bu durumda eşleşen bir imza olduğu anlamına gelir ve `True` değerini döndürür, aksi takdirde `False` değerini döndürür.

`on_complete` fonksiyonu, cuckoo imza sürecinin sonunda çağrılır. Diğer işlevler, belirli olaylarda önce çağrılacak ve daha sofistike ve hızlı imzalar yazmanıza yardımcı olacaktır.

İmza eşleşirse, global konteynerin "signatures" bölümüne yaklaşık olarak aşağıdaki gibi yeni bir giriş eklenir:

```
"signatures": [
    {
        "severity": 2,
        "description": "Creates a Windows executable on the filesystem",
        "alert": false,
        "references": [],
        "data": [
            {
                "file_name": "C:\\d.exe"
            }
        ],
        "name": "creates_exe"
    }
]
```

Yeni İmza Oluşturmak

Sizinle birlikte çok basit bir imza oluşturma sürecini daha iyi anlamanızı sağlamak için birlikte oluşturacağız ve adımları ile kullanılabilir seçenekleri inceleyeceğiz. Bu amaçla, analiz edilen kötü amaçlı yazılımın 'i_am_a_malware' adında bir mutex açıp açmadığını kontrol eden bir imza oluşturacağız.

İlk yapmanız gereken bağımlılıkları içe aktarmak, bir iskelet oluşturmak ve bazı başlangıç özniteliklerini tanımlamaktır. Şu anda ayarlayabileceğiniz öznitelikler şunlardır:

- name: imza için bir tanımlayıcı.
- description: imzanın neyi temsil ettiğine dair kısa bir açıklama.
- severity: eşleşen olayların ciddiyetini belirleyen bir sayı (genellikle 1 ile 3 arasında).
- categories: eşleşen olayın türünü tanımlayan bir kategori listesi (örneğin "banker", "enjeksiyon" veya "anti-vm").
- families: imzanın özellikle bilinen bir kötü amaçlı yazılım ailesiyle eşleşiyorsa, bir liste halinde kötü amaçlı yazılım aile adları.
- authors: imzayı oluşturan kişilerin listesi.
- references: imzaya bağlam sağlamak için referansların (URL'lerin) listesi.
- enable: False olarak ayarlanırsa imza atlanacaktır.
- alert: True olarak ayarlanırsa, imzanın raporlanması gerektiğini belirtmek için kullanılabilir (belki de özel bir raporlama modülü tarafından).
- minimum: Bu imzanın başarıyla çalıştırılabilmesi için gereken minimum Cuckoo sürümü.
- maximum: Bu imzanın başarıyla çalıştırılabilmesi için gereken maksimum Cuckoo sürümü."

Örneğimizde aşağıdaki iskeleti oluşturacağız:

```
from cuckoo.common.abstracts import Signature

class BadBadMalware(Signature): # We initialize the class inheriting Signature.
    name = "badbadmalware" # We define the name of the signature
    description = "Creates a mutex known to be associated with
Win32.BadBadMalware" # We provide a description
    severity = 3 # We set the severity to maximum
    categories = ["trojan"] # We add a category
    families = ["badbadmalware"] # We add the name of our fictional malware
family
    authors = ["Me"] # We specify the author
    minimum = "2.0" # We specify that in order to run the signature, the user will
simply need Cuckoo 2.0

    def on_complete(self):
        return
```

Bu, tamamen geçerli bir imzadır. Henüz bir şey yapmaz, bu yüzden şimdi imzanın eşleşmesi için koşulları tanımlamamız gerekiyor.

Dediğimiz gibi, belirli bir mutex adını eşleştirmek istiyoruz, bu nedenle şu şekilde devam ederiz:

```
```

```
from cuckoo.common.abstracts import Signature

class BadBadMalware(Signature):
    name = "badbadmalware"
    description = "Creates a mutex known to be associated with
Win32.BadBadMalware"
    severity = 3
    categories = ["trojan"]
    families = ["badbadmalware"]
    authors = ["Me"]
    minimum = "2.0"

    def on_complete(self):
        return self.check_mutex("i_am_a_malware")
```

Böylece, şimdi imzamız, analiz edilen kötü amaçlı yazılımın belirtilen mutex'u açılırken gözlemlenip gözlemlenmediğini döndürecektir.

Daha açık olmak ve doğrudan global konteynere erişmek istiyorsanız, önceki imzayı şu şekilde çevirebilirsiniz:

```
from cuckoo.common.abstracts import Signature

class BadBadMalware(Signature):
    name = "badbadmalware"
    description = "Creates a mutex known to be associated with
Win32.BadBadMalware"
    severity = 3
    categories = ["trojan"]
    families = ["badbadmalware"]
    authors = ["Me"]
    minimum = "2.0"

    def on_complete(self):
        for process in self.get_processes_by_pid():
            if "summary" in process and "mutexes" in process["summary"]:
                for mutex in process["summary"]["mutexes"]:
                    if mutex == "i_am_a_malware":
                        return True

        return False
```

Evented İmzalar

1.0 sürümünden itibaren Cuckoo, daha yüksek performanslı imzalar yazma olanağı sunmaktadır. Geçmişte, her imzanın analiz sırasında toplanan tüm API çağrıları koleksiyonu üzerinde döngü yapması gerekiyordu. Bu, bu tür bir koleksiyonun büyük boyutta olması durumunda gereksiz performans sorunlarına neden oluyordu.

1.2'den itibaren Cuckoo, yalnızca "evented imzaları" desteklemektedir. Eski run fonksiyonuna dayalı imzalar, `on_complete` kullanarak port edilebilir. Temel fark, bu yeni formatta tüm imzaların paralel olarak yürütülecek olması ve API çağrıları koleksiyonu üzerinde tek bir döngü ile her imza için `on_call()` adlı bir geri çağrı fonksiyonunun çağrılacak olmasıdır.

Bu tekniği kullanan örnek bir imza şudur:

```
from cuckoo.common.abstracts import Signature

class SystemMetrics(Signature):
    name = "generic_metrics"
    description = "Uses GetSystemMetrics"
    severity = 2
    categories = ["generic"]
    authors = ["Cuckoo Developers"]
    minimum = "2.0"

    # Evented signatures can specify filters that reduce the amount of
    # API calls that are streamed in. One can filter Process name, API
    # name/identifier and category. These should be sets for faster lookup.
    filter_processnames = set()
    filter_apinames = set(["GetSystemMetrics"])
    filter_categories = set()

    # This is a signature template. It should be used as a skeleton for
    # creating custom signatures, therefore is disabled by default.
    # The on_call function is used in "evented" signatures.
    # These use a more efficient way of processing logged API calls.
    enabled = False

    def on_complete(self):
        # In the on_complete method one can implement any cleanup code and
        #  decide one last time if this signature matches or not.
        #  Return True in case it matches.
        return False

    # This method will be called for every logged API call by the loop
    # in the RunSignatures plugin. The return value determines the "state"
    # of this signature. True means the signature matched and False it did not
```

```
this time.
    # Use self.deactivate() to stop streaming in API calls.
    def on_call(self, call, pid, tid):
        # This check would in reality not be needed as we already make use
        # of filter_apinames above.
        if call["api"] == "GetSystemMetrics":
            # Signature matched, return True.
            return True

        # continue
        return None
```

Satır içi yorumlar zaten kendi kendini açıklayıcıdır.

Bir imza eşleştiğinde başka bir olay tetiklenir.

```
required = ["creates_exe", "badmalware"]

def on_signature(self, matched_sig):
    if matched_sig in self.required:
        self.required.remove(matched_sig)

    if not self.required:
        return True

    return False
```

Bu tür bir imza, anormallikleri tanımlayan birkaç imzayı birleştirmek için kullanılabilir ve örneği sınıflandıran bir imza (malware uyarısı) oluşturabilir.

İşaretler ve Yardımcılar

1.2 sürümünden itibaren imzalar, imzayı tetikleyen şeyi tam olarak kaydetme yeteneğine sahiptir. Bu, kullanıcıların bu imzanın günlükte neden bulunduğunu daha iyi anlamalarına ve kötü amaçlı yazılım analizine daha iyi odaklanabilmelerine olanak tanır.

İşaretler ve yardımcılar konusundaki örnekler için şu anda Cuckoo Community'e başvurun.

Raporlama Modülleri

Raw analiz sonuçları işleme modülleri tarafından işlendikten ve soyutlandıktan sonra (bkz. Processing Modülleri), Cuckoo tarafından mevcut tüm raporlama modüllerine iletilir. Bu modüller, küresel konteynere erişir ve farklı formatlarda erişilebilir ve tüketilebilir hale getirir.

Başlarken

Tüm raporlama modülleri, `cuckoo/cuckoo/reporting/` dizini içine yerleştirilmelidir (bu, `cuckoo.reporting` modülüne çevrilir).

Her modül ayrıca `$CWD/conf/reporting.conf` dosyasında özel bir bölüme sahip olmalıdır: örneğin, `cuckoo/cuckoo/reporting/foobar.py` adında bir modül oluşturursanız, `$CWD/conf/reporting.conf` dosyasına (ve böylece Git deposundaki `cuckoo/data/conf/reporting.conf`) aşağıdaki bölümü eklemelisiniz:

```
[foobar]
enabled = on
```

Sizin bölümünüze eklediğiniz her ek seçenek, raporlama modülünüzde self.options sözlüğünde kullanılabilir olacaktır.

Çalışan bir JSON raporlama modülü örneği aşağıdadır:

```
import os
import json
import codecs

from cuckoo.common.abstracts import Report
from cuckoo.common.exceptions import CuckooReportError

class JsonDump(Report):
    """Saves analysis results in JSON format."""

    def run(self, results):
        """Writes report.
        @param results: Cuckoo results dict.
        @raise CuckooReportError: if fails to write report.
        """
```

```
        try:
            report = codecs.open(os.path.join(self.reports_path, "report.json"), "w",
"utf-8")
            json.dump(results, report, sort_keys=False, indent=4)
            report.close()
        except (UnicodeError, TypeError, IOError) as e:
            raise CuckooReportError("Failed to generate JSON report: %s" % e)
```

Bu kod basitçe, işleme modülleri tarafından üretilen küresel konteyneri alır, JSON'a dönüştürür ve bir dosyaya yazdırır.

Geçerli bir raporlama modülü yazmak için birkaç gereklilik vardır:

- `Report` sınıfından alan bir sınıfınızı bildirin.
- Ana işlemleri gerçekleştiren bir `run()` fonksiyonunuz olsun.
- Mümkünse çoğu istisnayı yakalamaya çalışın ve bir sorunu bildirmek için `CuckooReportError` istisnasını yükseltin.

Tüm raporlama modülleri, bazı özniteliklere erişime sahiptir:

- `self.analysis_path`: ham analiz sonuçlarını içeren klasörün yolu (örneğin, `storage/analyses/1/`)
- `self.reports_path`: raporların yazılması gereken klasörün yolu (örneğin, `storage/analyses/1/reports/`)
- `self.options`: `conf/reporting.conf` dosyasındaki raporun yapılandırma bölümünde belirtilen tüm seçenekleri içeren bir sözlük.