防调试和防篡改

防调试与防篡改是两种最常见的防操纵技术。调试器是进行动态逆向分析时常用的工具，防调试的主要目的是让软件难以被调试。防篡改的目的则是为了保证软件的完整性，避免软件某一部分功能被改变，或者软件的部分功能被提取出来独立运行。

本章会对几种代表性的防篡改与防调试技术进行原理性地介绍。

防调试

调试器可以监视、控制、甚至改变软件的运行过程，是动态逆向分析的常用工具。从被调试的目标软件上，调试器可以分为应用软件调试器和系统软件调试器。从调试器的使用方式上，可以分为本地调试器和远程调试器。从实现原理上，可以分为软件调试器和硬件调试器。

防调试的思路是根据调试的原理，对调试行为进行检查并加以限制。

在介绍防调试的方法前，我们要先看看调试器的原理。在今后的章节中，每当介绍一个防御技术时，我们都会首先介绍它要针对的是哪种攻击技术。这样才能理解该防御技术的用途和局限性。

调试器的原理

调试器是用来调试运行中的软件的。软件可能是直接在处理器上运行，也可能是在指令集模拟器(instruction set simulator，简称模拟器)上运行的。字节码软件(比如Java程序)和脚本软件(比如JavaScript程序、Bash脚本等)，是在虚拟机里或解释器里执行的，它们的调试相对特殊，所以就不在这里进行介绍了。

当软件在处理器上直接执行时，调试器有两种方法对其进行调试。一种是用调试器作为父进程来启动软件，或者附着(attach)到正在运行的软件上。另一种办法是利用硬件的调试端口进行调试。后一种方法主要用作调试系统软件，或对嵌入式系统进行调试，且依赖在线仿真器(in-circuit emulator)进行。这里我们主要关注第一种调试方法。

当调试一个程序时，最常见的操作是设置断点。常见的断点设置方式有两种，一是利用软件自陷(trap)机制，修改想要设置断点位置处的质量为自陷指令，并注册中断处理函数。当软件执行到自陷指令时，就会把控制权交给操作系统并进入中断处理函数从而返回调试器。另一种方式是设置硬件断点。因为受到硬件资源限制，硬件断点的数目都是有限的。常见的处理器都有一组寄存器作为硬件断点使用，寄存器的值代表了要断下的地址。处理器会在执行到这些地址时自动暂停被调试的软件，并唤醒调试器。

虽然调试器种类众多、技术特点也各不相同，但针对一个具体的软件，总有一款或几款最便于对其进行调试的调试器。进行防调试时，只需要针对这些热门调试器进行防护即可。这样会逼迫攻击者要么使用不那么方便的调试器，要么(如果有条件的话)修改调试器甚至操作系统绕过防调试机制，要么只能修改被调试的软件来拆除防调试功能。

利用进程信息进行防调试

因为软件调试器一般都是被调试程序的父进程。所以可以利用这一点进行防调试。

可以让软件来调试自身，如果无法调试，则说明有调试器在调试软件。也可以通过调用系统的应用可编程接口(API)或者检查进程中的一些数据结构来检查软件是否在被调试。比如在Windows上，可以调用 `IsDebuggerPresent` 接口来进行检查。还可以通过进程名(或窗口名)检查用户是否启动了调试器。具体的防调试方法同操作系统有很强的相关性，所以我们就不展开讲了。

> 值得注意的是，这些防调试机制都可以被攻击者破坏。破坏防调试机制的技术统称为反防调试技术。比如说，攻击者可以通过静态逆向分析定位到关键的API调用，并通过篡改二进制代码的方式直接把调用指令删除，并添加指令将原本的返回值(通过修改寄存器或栈)改为伪造的结果。攻击者也可以修改操作系统，使得对某API调用时返回伪造的值，来欺骗防调试机制。

利用断点机制进行防调试

因为软件断点的实现基于对软件进行修改，所以可以对软件完整性进行检查，通过检查软件是否被篡改来检查自身是否在被用软断点的方式调试。完整性检查的具体做法在本章的防篡改部分会展开介绍。

还可以反其道而行之，在软件自身代码里提前放置好一些自陷指令，且不提供中断处理函数，而是用异常处理机制捕捉执行到自陷指令后引发的异常。这些异常都是期望中应该被执行到的，如果没有执行到，则说明软件在被调试。

硬件断点的机制是通过设置相关寄存器完成的，所以也可以通过检查这些寄存器的值来得知软件是否在被用硬件断点的方式调试。

> 如果这类防调试方法做的足够好，则攻击者一般只能通过修改软件的方式来切除防调试功能。所谓切除，是通过对二进制软件进行篡改完成的。切除的方法有很多，比如将防调试相关的指令删除掉，或者用跳转指令跳过防调试逻辑。当然，在软件经过大量混淆，且攻击者无法很方便地对软件进行调试的情况下，切除防调试功能也并非易事。

通过检查程序运行状态来防调试

防御者也可以通过其他维度的信息来间接判断软件是否在被调试。比如可以记录某段代码的执行时间，如果耗时异常地长，则可能有攻击者在调试软件。

值得注意的是，这类检查方法往往有较高的误报率(false-positive rate)，即误以为软件在被调试，所以应该慎用。

对抗特定的调试器

有些流行的调试器本身有漏洞或缺陷。可以构造软件触发这些缺陷，造成调试器无法正常工作。这种方法只对某些特定版本的调试器有用，所以有时效问题。

防篡改

一方面，在很多情况下，软件逆向的目的就是对软件进行篡改。比如对软件进行盗版的方式一般就是通过篡改软件破坏软件的授权检查逻辑。

另一方面，通过篡改软件，可以更方便地对软件进行逆向。比如软件中的防调试逻辑是软件逆向的一个阻碍，这个阻碍就可以通过篡改软件的方式解除。

由于这些原因，防篡改成为软件保护中非常重要的技术。

完整性校验

要防篡改，最直接的思路是对软件进行完整性校验，从而检测出篡改，并对篡改做出响应，比如人为制造程序奔溃。

下面我们用一个例子展示完整性校验。

代码 1.原始代码

```c
int license_expired() {
    struct timespec ts;
    if (clock_gettime(CLOCK_REALTIME, &ts) != 0) {
        // Failed to get time.
        return 1;
    }
    if (ts.tv_sec > 1577836800) {
        // Expired after Jan 1st, 2020.
        return 1;
    }
    return 0;
}

int main() {
    if (license_expired()) {
        printf("Error: License expired!\n");
        return 1;
    }
    // Do other jobs.
    return 0;
}
```

这是一段C程序。作者用硬编码(hardcoding)的方式让程序在2020年1月1日后授权过期并退出运行。(程序中出现的 `1577836800` 是2020年1月1日0点0分0秒所对应的标准时间。)

这段程序的实现非常天真，破解的方法非常多，比如攻击者只要把进行比较的 `>` 操作改成 `<` 就可以让程序的检查逻辑反转，从而在2020年后让软件继续运行。如何做到修改比较操作呢？攻击者可以通过调试的方法定位到进行该操作的比较指令，并将其改成与其逻辑相反的指令。比如在x86处理器上，如果编译器生成的指令是 `JG` (jump if greater than)指令改成 `JL` (jump if less than)指令。这两个指令的操作码(opcode)分别是 `0x7F` 和 `0x7C` ，所以只用修改程序里的两个比特就可以反转检查逻辑了。

为了避免程序被轻易修改，开发者可以在程序中引入完整性校验逻辑。比如下面代码中的 `license_expire` 函数内就添加了完整性校验逻辑。

代码 2.完整性校验1

```
1  // The value of reference_value should be modified after compilation.
2  unsigned char reference_value = 0xFE;
3
4  int license_expired() {
5      unsigned char hash;
6      unsigned char * p;
7      struct timespec ts;
8      // Code integrity check.
9      hash = 0;
10     for (p = (unsigned char *)&&begin;
11          p < (unsigned char *)&&end;
12          ++p) {
13         hash ^= *p;
14     }
15     if (hash != reference_value) {
16         printf("The code is tampered!\n");
17         exit(1);
18     }
19 begin:
20     if (clock_gettime(CLOCK_REALTIME, &ts) != 0) {
21         // Failed to get time.
22         return 1;
23     }
24     if (ts.tv_sec > 1577836800) {
25         // Expired after Jan 1st, 2020.
26         return 1;
27     }
28 end:
29     return 0;
30 }
31
32 int main() {
33     if (license_expired()) {
34         printf("Error: License expired!\n");
35         return 1;
36     }
37     // Do other jobs.
38     return 0;
39 }
```

这样，每当 `license_expired` 函数执行时，第7行到第17行间的完整性校验逻辑就会校验从标号 `begin` 到 `end` 之间的代码的完整性。校验方式是计算目标代码的哈希(hash)值。这里用到了GCC引入的扩展C语法 `&&` 来取一个标号对应的地址。目标代码的计算出的哈希值可以同期望的标准哈希值(例子中的 `reference_value`)进行比较，从而实现检查目标代码的内容有没有被篡改的目的。

这里有一个有趣的问题：我们怎么知道标准哈希值是多少呢？事实上，由于编译的复杂性，除非编译(甚至链接)

一个程序，否则我们是无法获知某段代码的哈希值的。这样我们就需要先编译程序，对编译后的程序进行分析，从而计算出标准哈希值，再修改编译出的程序，把标准哈希值填入程序中。值得注意的是，必须保证填入标准哈希值的过程不会影响到目标代码。在这个例子里，我们把标准哈希值存在一个全局变量里，所以可以通过修改二进制程序的数据段(data section)来改变标准哈希值，从而避免填入的过程影响目标代码。

进行完整性校验的逻辑一般叫做检测器(tester)。一个程序中可以有多个检测器，每个检测器检测一段目标代码。必须说明的是，常规的检测器是无法检测自身的。我们可以修改检测器使其检测自身的部分代码，但一般无法检测所有的代码。比如下面的修改过的代码里，检测器的目标代码的范围看起来涵盖了检测器自身，但其实由于 `reference_value` 没有受到检测，所以检测器还是没有检测自身的完整性。

代码 3.完整性校验2

```
// Not included in the integrity check.
unsigned char reference_value = 0xFE;

int license_expired() {
    unsigned char hash;
    unsigned char * p;
    struct timespec ts;
begin:
    // Code integrity check.
    hash = 0;
    for (p = (unsigned char *)&&begin;
         p < (unsigned char *)&&end;
         ++p) {
        hash ^= *p;
    }
    if (hash != reference_value) {
        printf("The code is tampered!\n");
        exit(1);
    }
    if (clock_gettime(CLOCK_REALTIME, &ts) != 0) {
        // Failed to get time.
        return 1;
    }
    if (ts.tv_sec > 1577836800) {
        // Expired after Jan 1st, 2020.
        return 1;
    }
end:
    return 0;
}
```

如果如下面的代码所示对检测器的实现进行修改，把 `reference_value` 的值也纳入检测中，则会出现鸡生蛋蛋生鸡的逻辑悖论。由于 `reference_value` 也参与了标准哈希值的计算，当我们把计算出的标准哈希值写入 `reference_value` 后，实际计算出的哈希值就会变化，本来的标准哈希值就不再正确了。

代码 4.完整性校验3(错误的做法)

```
// Included in the integrity check,
// but its value can no longer be modified.
unsigned char reference_value = 0xFE;

int license_expired() {
    unsigned char hash;
    unsigned char * p;
    struct timespec ts;
begin:
    // Code integrity check.
    hash = 0;
    for (p = (unsigned char *)&&begin;
         p < (unsigned char *)&&end;
         ++p) {
        hash ^= *p;
    }
    hash ^= reference_value;
    if (hash != reference_value) {
        printf("The code is tampered!\n");
        exit(1);
    }
    if (clock_gettime(CLOCK_REALTIME, &ts) != 0) {
        // Failed to get time.
        return 1;
    }
    if (ts.tv_sec > 1577836800) {
        // Expired after Jan 1st, 2020.
        return 1;
    }
end:
    return 0;
}
```

虽然上面这段代码存在逻辑悖论问题，但我们可以看到，完整性校验不只可以校验指令代码，也可以对数据进行校验。

完整性校验的弱点

了解了完整性校验的原理后，我们可以开始分析这种技术的弱点。

- 首先，完整性校验并不参与软件的原本功能的实现。也就是说，完整性校验是实现了一些额外的功能。这些额外的功能如果被切除掉，软件原本的功能不会受到影响。事实上攻击者的攻击方式一般也经常是切割掉完整性校验。值得一提的是，前面一节介绍的防调试也有同样的弱点，这也是为什么对防调试的攻击也往往是切除防调试功能。
- 其次，完整性校验逻辑一般有比较明显的特征。比如有的校验算法会用到异或指令，校验逻辑一般会读取代码段的内容等。

- 再次，一个检测器一般不会校验很多的代码。因为校验的代码越多，性能开销就越大，完整性校验的逻辑也越容易被定位到。
- 最后，为了减少性能开销，检测器的运行频率也不会过高。一般检测器会在执行被检测的目标代码前一段时间执行。如果检测器和目标代码的执行离得太近，检测器就更容易暴露。如果离得太远，攻击者进行的动态篡改就更难被发现。比如说，当攻击者设置软件断点时，其实就是对软件进行了篡改，如果在执行完检测器后对目标代码设置断点，则检测器将没有机会检测到断点所造成的篡改。

既然完整性校验有这些弱点，攻击者就可以进行针对性地攻击。

基本攻击手段

对完整性校验进行攻击的手段很多，这里我们主要介绍一下基本的手段。

- 篡改检测器，主要方式如下：
  - 篡改标准哈希值。由于检测器无法保护自身，尤其是难以保护标准哈希值，所以这是最自然的攻击手段。然而这种手段缺少灵活性，其使用并不常见。
  - 篡改检测逻辑，使其无法正常工作。比如，攻击者可以用类似于前面介绍的把比较指令 `JG` 改为 `JL` 的方式，来把完整性校验2 这个例子里对应 `if (hash != reference_value)` 语句而生成的 `JNE` (jump if not equal to)指令改为 `JE` (jump if equal to)指令。这样即使这块代码的哈希值产生了变化，由于检查逻辑本身已经被反转，所以会(几乎)总是通过。
- 绕过检测器。检测器也是要执行的，但它们不属于软件有效功能的一部分。所以如果能够彻底不执行检测器，就可以避免完整性校验。绕过检测器的难度取决于检测器的实现方式，在我们的例子里，可以把检测器开头的代码篡改为绝对跳转指令，跳过检测器；当检测器是用单独的函数来实现时，则绕过的方法更为简单。
- 欺骗检测器。由于检测器并不总是在运行，所以可以在检测器结束运行后对代码进行篡改，而在检测器再次运行前将被篡改的代码恢复成原始值。

改进型的完整性校验

在了解完整性校验的概念和基本攻击手段后，我们来看一下改进型的完整性校验的设计。

增加检测算法的多样性

篡改检测器和绕过检测器的前提是定位到检测器。我们可以增加检测算法的多样性，来加大定位检测器的难度。这主要体现在哈希算法的多样性上。由于用在检测器里的哈希算法对低碰撞概率的要求并不高，所以现实中可以选择的算法很多。比如上面例子中的算法可以改变成用更隐蔽的指令实现：

代码 5.另一种哈希算法

```
    for (p = (unsigned char *)&begin;
         p < (unsigned char *)&end;
         ++p) {
        hash += *p * 777;
    }
}
```

混淆内存读取特征

即使改变了检测器的哈希算法，检测器还有一点明显特征，即会读取代码段(text section)的内容。有心的攻击者可以先用模拟器运行一遍软件，记录下所有对内存的读操作，并通过读代码段的特征识别检测器。

为了对抗这种识别方法，防御者可以进行两种迷惑操作：

- 首先，可以把正常情况下应该存储在数据段中的数据存储在代码段。这样软件里原本不属于检测器的代码就也会读取代码段的内容。
- 另外，可以把本来要执行的代码用数据化的方式存储。这种技术主要是利用虚拟机的原理实现的，我们会在第五章对虚拟机混淆进行介绍。由于代码存储在了数据段，检测器的行为就变为读取数据段的内容了，这样检测器自然更难被识别。

功能融合

为了避免检测器被绕过，可以把软件普通的功能混入检测器中，或者让检测器的检测结果参与软件的正常功能。比如，检测器计算出的哈希值不是和标准哈希值进行比较，而是作为偏移量用来索引某些数据。比如下面的例子里，假设 `checksum` 函数是一个检查器，用来计算某段目标代码的哈希值，算好的值可以参与到运算中来。如果目标代码被修改了，则算出的错误哈希值会造成取变量 `g` 的值的函数 `get_g` 返回错误的值，影响软件的普通功能。当然，这个例子既可以理解为检测器参与软件的普通功能，也可以理解为发生篡改后用破坏软件功能进行反制。我们将在第三节从反制策略的角度介绍防调试和防篡改。

代码 6.把哈希值用在计算中

```
unsigned char reference_value = 0xFE;

// A global variable.
int g;

// Get the value of g.
int get_g() {
    return *(g + checksum() - reference_value);
}
```

多层校验

为了避免标准哈希值或检测逻辑遭到篡改，我们可以在程序中添加一些额外的检测器，来检测其它检测器。这样就形成了双层校验。可以继续把这个概念外推，形成多层校验。这样检测器之间会形成有向无环的依赖关系。

当然，最外层的检测器是可以被篡改的，从这里入手，攻击者仍然可以逐层破解校验逻辑。

进阶攻防

针对防篡改，还有一些进阶的攻击手段。这里我们介绍两种攻击手段及相应的防守方法。

双子攻击

Philippe Biondi和Fabrice Desclaux在BlackHat欧洲会议上介绍了对Skype的逆向分析。为了对Skype进行调试，他们使用了双子攻击的手段破解Skype的防篡改功能。

> 进行双子攻击，首先需要定位到所有的检测器。然后可以同时启动两个软件进程。在一个软件进程里，攻击者在每个检测器的调用点设置软件断点。显然，由于设置软件断点会造成对软件的篡改，此时一些检测器将能够检测出篡改并报错。
>
> 攻击者先在第一个进程里进行操作，当进程在某个软件断点停下来时就意味着即将执行某个检测器了。此时攻击者在第二个进程对应于该检测器执行结束的位置设置一个硬件断点，并在第二个进程里执行与第一个进程相同的操作。由于第二个进程没有被设置软件断点，所以检测器会计算出正确的哈希值。在第二个进程停在硬件断点后，攻击者把获得的哈希值拷贝到第一个进程里，并跳过第一个进程里的检测器。用这种办法，攻击者可以把所有检测器都切割掉。

双子攻击的前提是当对两个进程执行相同的操作时，两个进程的执行路径一致。所以可以通过对功能进行随机化进行对抗。

修改操作系统或模拟器

当攻击者只需要篡改代码段的内容时，可以用修改操作系统或模拟器的方式进行攻击。

> 对代码段的读取会触发操作系统的缺页中断，所以可以修改操作系统，当缺页中断读取代码段时，让操作系统读取未经修改的软件中相应的内存页。这样检测器会永远读到未经修改的代码段的内容，但却执行修改了的代码段的内容。防篡改逻辑就自然失效了。
>
> 另一种类似的办法是将软件在模拟器里执行，修改模拟器，当装载指令(load instruction)读取代码段时，返回未经修改的软件中相应的内存的值。这样的效果和修改操作系统是一样的。但由于软件在模拟器里执行，且模拟器需要监视每一次装载操作，所以性能开销很大。

这两种攻击方法都有较大的局限性。首先，这类攻击方法只能在攻击者自己的设备上运行。攻击者无法篡改软件并发布给其他人。其次，如果使用了虚拟机混淆的方式把代码进行了数据化处理，则攻击者就很难通过只修改代码段来完成想要进行的篡改。

间接防篡改

除了对程序进行完整性校验，还可以通过检查程序的运行状态来实现防篡改。检查的方法很多，比如如下的代码里，第24行的语句是不应该执行到的，当它被执行时，即可认为程序被篡改了。攻击者可能跳过了 `license_expired` 的执行，或者在执行过后篡改了返回值。我们看到不管是执行过程被篡改还是数据被篡改，都会造成 `g_is_expired` 的值出现异常。

代码 7.利用冗余数据间接防篡改

```c
int g_is_expired = 1;

int license_expired() {
    struct timespec ts;
    if (clock_gettime(CLOCK_REALTIME, &ts) != 0) {
        // Failed to get time.
        return 1;
    }
    if (ts.tv_sec > 1577836800) {
        // Expired after Jan 1st, 2020.
        return 1;
    }
    g_is_expired = 0;
    return 0;
}

int main() {
    if (license_expired()) {
        printf("Error: License expired!\n");
        return 1;
    }
    // When the program reaches here, g_is_expired should always be 0.
    if (g_is_expired) {
        printf("Error: Program tampered!\n");
        return 1;
    }
    // Do other jobs.
    return 0;
}
```

从原理上讲，这种防篡改所利用的是数据的冗余性。如果数据没有被完整地篡改，软件就有机会检查出自身执行异常，从而可以采取反制措施。

防黑盒调用与防重打包

黑盒调用与重打包本质上也是篡改行为。防黑盒调用与防重打包也属于防篡改的范畴。

所谓黑盒调用，指的是把软件看做一个黑盒子，从外部调用作者想要禁止他人使用的接口。黑盒调用常常用在调用库(library)函数上。很多时候，软件是以库文件的形式存在的。比如说，在安卓应用中，机器指令实现的部分必须以共享库(shared library)文件(常以.so为后缀)的形式存在。如果应用的作者想要避免盗版者直接把自己的库文件拿去使用，就必须采用一些防篡改措施。

防黑盒调用的具体做法很多，但都可以归为两类机制。一类机制基于由被调用者检查调用者的信息。比如在安卓平台上，共享库可以检查应用的bundle信息等。另一类机制基于问答，有点像接头暗号。此时调用方法会相对复杂。下面的代码展示了一个基于问答的例子。调用者(`caller`)调用被调用者(`callee`)时需要传递一个回调函数(`callback`)，被调用者可以构造随机输入，验证回调函数执行结果是否符合预期。如果不符合预期，则证明

有其它软件调用了被调用者。为了对抗这种防黑盒调用机制，攻击者需要对回调函数进行逆向，而回调函数可以设计的很复杂，且用代码混淆技术进行保护，从而增大攻击难度。

代码 8. 调用者(caller)

```
1 /**** Caller File ****/
2 int callback(int a, int b) {
3     return a % b;
4 }
5
6 void caller() {
7     callee(callback);
8 }
```

代码 9. 被调用者(callee)

```
 1 /**** Callee File ****/
 2 typedef int cb(int a, int b);
 3 void callee(cb callback) {
 4     int a = random();
 5     int b = random();
 6     int result = callback(a, b);
 7     if (result != a % b) {
 8         printf("Error: Wrong caller!\n");
 9         return;
10     }
11     // Do some job.
12 }
```

所谓重打包，指的是对软件进行逆向后，替换掉软件的一部分内容，再重新生成新的软件。要防重打包，一方面可以靠代码混淆提升逆向难度，另一方面可以用防篡改技术校验软件可能被替换的部分的完整性。

反制措施

在前面的例子里，当发现软件遭到调试或篡改，我们一般都打印一个错误，并让软件结束运行。例子里的实现都过于简单，目的是为了演示方便。事实上如果这么做，攻击者很容易根据报错信息反向定位到防调试或防篡改逻辑并对其发动攻击。

在本节，我们对常见的反制措施进行介绍。

报错

最常见的反制方式是报错。报错的用户体验是最好的。在一些场景下，比如授权过期，为了不对正常过期用户造成困扰，软件只能用报错的方式进行提示。对于防调试和防篡改而言，是否需要采用报错的方式取决于软件的目标用户与运行场景，不能一概而论。

如果检测到调试或篡改行为后进行报错，应该充分分离报错代码与检测代码，避免攻击者顺藤摸瓜，从报错逻辑

回溯定位到检测器。所谓分离，包括三个方面：

- 一、检测代码和报错代码之间应该经历足够久的时间、执行足够多的其它指令。这即**时间分离**。
- 二、检测代码和报错代码在软件内的地址应相隔足够远。即**空间分离**。
- 三、检测代码需要向报错代码传递信息(比如使用变量标记错误)来控制报错代码的执行。这个信息应该足够晦涩，经历足够多的处理。这也叫做**逻辑分离**。

其中，逻辑分离非常重要，但往往不太受重视。做逻辑分离时可以用代码混淆的方式使得反向跟踪数据的传播过程难以进行，这样的代码混淆我们会在第四章介绍。对于具有客户端服务端结构的软件而言，还可以由检测代码向服务端发送隐蔽的信息，并由服务端再通知客户端进行报错，从而做到更好的逻辑分离。

崩溃

相比于报错，更隐蔽的反制措施是人为制造程序崩溃或其它运行错误。为了防止回溯，崩溃点也需要和检测代码做到时间分离、空间分离和逻辑分离。除此以外，还有一些额外的技巧可以更好地防回溯，这里举两个例子。

- 制造随机崩溃。程序可以维护一个可以造成崩溃的变量列表。包括但不限于指针、偏移量等用来控制访问内存地址的变量。当检测器发现软件被篡改后，可以修改任意一个或多个变量，从而制造出将来某一时刻的崩溃。
- 破坏函数栈(stack)。还有一个技巧是破坏掉栈上的一些关键数据，制造错误或崩溃。比如，通过修改当前帧(frame)的返回地址，可以控制当前函数返回后不回到本该执行的指令上，从而造成错误，甚至引发崩溃。破坏栈时可以破坏距离当前函数较远的帧。此时关键数据的位置可以通过对栈进行回溯(backtracking)分析的方法得知。回溯分析在源码级调试器(source-level debugger)中应用广泛，感兴趣的读者可以自行分析GDB里 `bt` 命令的实现。破坏栈时还可以同时对多个帧进行破坏，这样能够有更大的把握让程序崩溃，且程序崩溃后攻击者更难定位崩溃原因。

上报

我们前面已经讲到，对于具备客户端服务端结构的软件而言，可以通过向服务端上报，并由服务端再通知客户端进行反制，从而实现较好的逻辑分离。除此以外，风险信息上报还有许多其它用途，这里列举一二。

- 服务端接收到风险信息后，可以将信息以及于该客户端关联的设备信息和账号信息记录下来，将来再进行限制或惩罚。比如说，黑灰产通过篡改电商类客户端软件可以进行机器行为抢红包等活动作弊，此时黑灰产往往通过使用VPN、拨号上网换IP等方式保护自己免于被溯源。但黑灰产在尝试篡改客户端的时候经常不注意进行保护，服务端在收到风险信息后可以不立即反制，而是记录下各种信息，等黑灰产正式开始作弊后利用此前收集到的信息进行溯源。
- 当发现软件被篡改后，客户端除了可以把风险通知服务端外，还可以把被篡改的软件隐蔽地传到服务端，从而让防御者有机会知道攻击者是如何对软件进行篡改的。为了隐蔽性，当被篡改的部分较多时，客户端可以把其中随机位置的一小段代码传到服务端。如果被篡改的客户端投入了批量使用，服务端就可以利用每个客户端传回的小段信息拼接出完整的被篡改的代码。

其它

除了这些措施外，还有其它不太常见的反制措施。其中一种反制措施是对软件进行修复，当攻击者修改了软件中的某些代码或数据时，由于软件提前进行了冗余存储，所以可以把被篡改的内容恢复回去。还有一种措施是减少软件的功能或降低性能，这个措施可以用在防盗版上，目的是使盗版软件表现不如正版软件，比如微软就曾经通

过让盗版的Windows操作系统壁纸变为纯黑背景加盗版文字提示，来轻微破坏盗版软件的用户体验。

实现与应用

本章前面各节阐述了防调试与防篡改的原理，下面我们分析一下如何实现这些技术。

实现方法

本章介绍的所有技术都可以用手工的方式进行实现。然而，手工实现容易出错，且模式往往较为单一，所以并不值得推荐。

另外，防调试与防篡改本身都应该用代码混淆进行进一步保护，因为防调试与防篡改代码本身就是攻击者进行分析和篡改的目标。

考虑到这些现实问题，我们推荐软件保护工程师用工具化的方法完成防调试和防篡改。实现工具化的做法有很多种，这里我们介绍三种可行的流程。

源码级实现

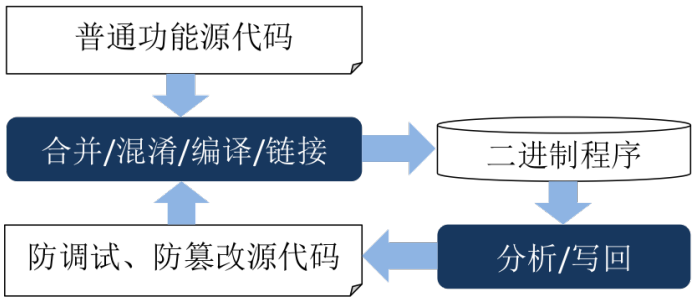

Figure 1. 源码级实现的流程

在这种流程中，软件开发者需要提供实现普通功能的源代码和负责防调试、防篡改的源代码。这些源代码需要合并到一起。可以用手工的方式在普通源代码中插入对防调试、防篡改源代码的调用。更好的做法是开发自动化的工具将防调试、防篡改源代码自动插入到普通源代码中。

合并后的源代码经过混淆、编译、链接后生成二进制程序，再由分析器将二进制程序中所有要进行完整性校验的目标代码的标准哈希值计算出来，最后把标准哈希值写回防篡改源代码中。重复性地运行一遍合并、混淆、编译、链接流程，生成最终的二进制程序。

这里需要注意三点：

- 合并、混淆、编译、链接的过程必须是严格确定性的(deterministic)。也就是说，相同的输入必须能每次产生完全相同的输出。这样才能避免分析出的标注哈希值失效。
- 合并、混淆、编译、链接的次序并不是一成不变的。比如说，在编译的过程中进行合并和混淆也是一种办法。
- 标注哈希值写回源代码后，再次生成二进制代码时，被检测的目标代码不应受到任何影响。这需要很多技巧。比如在例子完整性校验1里，`reference_value` 的初值不能是 `0`，否则它会被编译器放到用来存储未初始化的数据的BSS段里。再将 `reference_value` 修改为(非零的)标准哈希值后，该变量会被存入数据段，造成地址改变，从而可能对目标代码造成影响。

- 如果使用多层校验，则这个流程需要迭代多次。从修改最外层检测器所用到的标准哈希值开始，逐步推进到最内层。

二进制实现

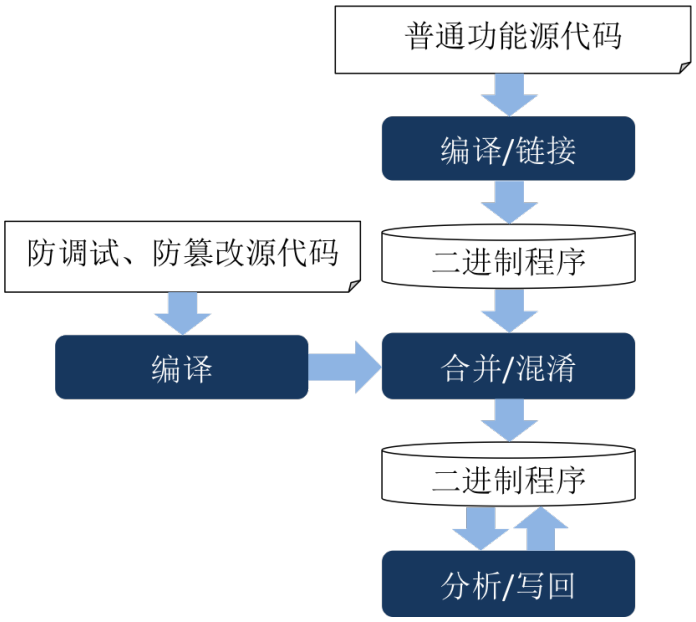

Figure 2. 二进制实现的流程

另一种实现方法是如上图所示的二进制实现。普通源代码经过编译和链接后生成二进制程序，然后所有的后续操作都是在二进制程序上完成的。防调试、防篡改代码经过编译后，可以和二进制程序进行合并与混淆。此后对二进制程序进行分析，获得标准哈希值，再把标准哈希值直接写回到二进制程序中去。

同源码级实现类似，当采用多层校验技术时，分析和写回标准哈希值的过程也需要迭代多次。

混合实现

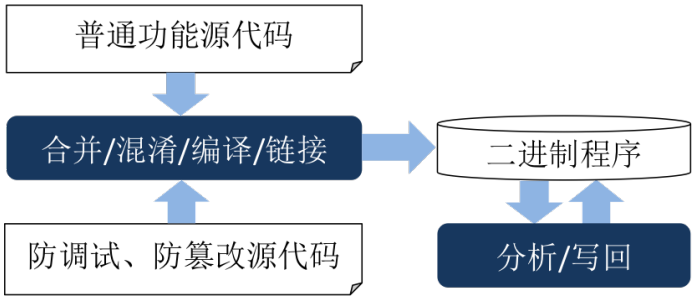

Figure 3. 混合实现的流程

最后，还可以结合源码级实现与二进制实现进行混合实现。混合实现同源码级实现很接近，唯一的区别是分析出的标准哈希值是直接写回二进制程序里的，而不用写回源代码。

比较

上面介绍的三种实现方法各有优缺点。我们不能孤立地看防调试和防篡改，而是必须结合代码混淆一起研究。

正如我们将来要介绍的，代码混淆主要有三种实现方法：一、在源代码上直接进行，二、在编译的阶段进行，三、在二进制程序上进行。第三种实现方法的用户体验最好，但能达到的混淆效果较差。

前面介绍的源码级实现和混合实现均可以较好地兼容这三种代码混淆方法，只要保证经过混淆后还能从二进制程序中分析出目标代码的位置。混合实现还要求在分析出标准哈希值后能找到需要写回二进制代码的位置。

二进制实现具有最好的用户体验，因为普通功能的开发者只需要使用现有开发流程生成二进制程序即可，此后可以将二进制程序提交给软件保护者或SaaS(Software-as-a-Service)化的软件保护服务平台，并最终获得一个受到保护的二进制程序。合并、混淆、分析、写回等过程相当于放在了一个黑盒子里，对于普通开发者都是透明的。 然而，这种实现只能使用效果较差的二进制混淆。

事实上，商业化的工具多采用二进制实现，而企业内部工具多采用源码级实现或混合实现。

应用策略

虽然防调试和防篡改的方法很多，但理论上这些技术都是可以被攻破的。引入越多的防调试和防篡改逻辑，就意味着软件在性能、尺寸、内存占用等方面有越大的开销。使用这些技术还可能对软件稳定性造成影响，并面临着影响用户体验的问题。尤其值得一提的的软件稳定性问题。引入任何新的代码，都可能对软件稳定性造成影响。然而防调试与防篡改代码应对的是软件被攻击的异常情况，构建测试用例对这些异常情况进行全面的测试是非常困难的，甚至有些时候是不可能做到的。所以在现实中，商业软件采用防调试和防篡改技术时通常比较谨慎。由于这些原因，我们不认为防调试与防篡改是多多益善的。

但是，应用多种保护技术的好处也是显而易见的。在应用防调试和防篡改技术时，也应配套使用代码混淆技术，否则防调试和防篡改逻辑本身也易于遭到攻击。另外，防篡改与防调试也是相辅相成的。

- 首先，调试和篡改可以以多种形式结合起来用来进行攻击，比如本章介绍的双子攻击就是一个例子。只有防调试和防篡改能力都得到增强才能抵御这种攻击。
- 其次，可以通过防篡改来提供部分防调试能力，如基于软件断点检测的防调试功能。
- 再次，许多调试器是可以修改被调试软件的运行状态的，所以可以被用作软件篡改工具。因此，防调试对防篡改也能提供一定的帮助。

综合来看，我们主张在尽量应用多种防护技术，同时合理控制每种技术的使用规模。

小结

本章介绍了防调试和防篡改的原理。防调试和防篡改的技术多种多样，这些技术各自能够针对一些攻击方式进行防御，但也都有相应的破解办法。从比喻的角度看，我们可以把攻击技术看做捕食者(preyer)而把保护技术理解为猎物(prey)。猎物都有一些自我保护的手段，可以让一些捕食者难以下口，但也各自有自己的天敌。

要实现防调试和防篡改，最好用工具化的办法完成。进行工具化的办法有很多，软件保护工程师应根据现实情况选择最适合的做法。

使用防调试和防篡改可能对用户体验、稳定性、软件性能、大小、内存开销等造成影响，所以要避免过度使用。我们建议尽量使用多种技术，但同时合理控制每种技术的使用规模。