

Baymax Patch Tools 帮助文档

# 劫持补丁全攻略

Ver 1.5

2023 年 1 月

# 目录

一. 工具概述.....	3
(一) 软件介绍.....	3
(二) 适用范围.....	3
(三) 如何检测.....	3
(四) 免责声明.....	4
二. 创建补丁项目 .....	4
(一) 设置目标进程 .....	4
(二) 添加补丁项并设置补丁进程和模块 .....	5
(三) 设置补丁地址 .....	5
1. 偏移补丁 .....	5
2. 中断补丁 .....	7
(四) 补丁加壳的程序.....	8
1. 使用 HOOK 模式 (推荐) .....	8
2. 设置中断模式 .....	9
3. 设置延时模式 .....	10
三. 搜索替换补丁 .....	10
(一) 偏移补丁.....	10
(二) 特征码替换补丁.....	11
四. 异常处理补丁 .....	13
(一) 设置中断地址 .....	13
(二) 中断类型.....	13
(三) 条件断点.....	14
1. 判断第几次中断 .....	14
2. 判断寄存器数值 .....	14
3. 判断寄存器指向的内存数值 .....	15
4. 通过寄存器偏移定位内存地址 .....	15
(四) 存储数据.....	17
(五) 修改寄存器.....	19
(六) 修改标志寄存器.....	19
(七) 修改寄存器指向的内存.....	20
(八) 修改 EIP/RIP .....	22
(九) 制作内存注册机.....	23
(十) 修改存储地址指向的内存.....	24
(十一) 添加反调试检测 .....	25
(十二) 修改函数的返回值 .....	25
1. 函数调用约定 .....	26
2. 三种修改函数返回值模式.....	27
3. 更好的理解任意指令均位于函数体内 .....	29
4. 任意栈地址返回后再进行补丁 .....	29
(十三) 对数据进行运算.....	29
(十四) 从 INI 文件中读取补丁信息.....	30
五. 生成补丁和保存补丁工程.....	31

(一) 菜单设置项说明.....	31
(二) 创建劫持、注入补丁 .....	33
(三) 创建调试版补丁.....	34
(四) 劫持模块和破解模块 .....	34
(五) 保存和打开补丁工程.....	35
六. 插件介绍.....	36
(一) PE 文件二进制比较 .....	36
(二) 特征码搜索工具.....	37
(三) 地址格式转换工具 .....	38
(四) 字符串转十六进制工具.....	38
七. 案例及使用技巧.....	39
(一) 通过 HOOK 实现对其他线程设置硬件断点 .....	39
(二) 通过对函数 HOOK 实现固定参数或返回值 .....	39
(三) 通过存储数据来补丁堆空间 .....	41
(四) 修改指令中的全局变量值.....	41
(五) 通过 HOOK 固定硬盘信息.....	43
八. 交流反馈.....	46
(一) 如何分析补丁失败原因.....	46
1. 请先关闭调试器 .....	46
2. 分析调试补丁的 LOG .....	46
3. 劫持模块加载时机过晚 .....	47
(二) 加入官方交流反馈群 .....	47
(三) 工具下载.....	47
(四) 更新日志.....	47

## 一. 工具概述

### （一）软件介绍

Baymax Patch Tools（简称 Baymax）通过对目标进程释放的劫持 DLL 来加载功能模块 PYG.DLL/PYG64.DLL，实现对目标进程的动态补丁。Baymax 不仅支持动态修改目标模块的指令和数据，还接管了进程 r3 的异常处理，模拟了调试器的异常处理功能，支持对目标模块地址设置断点，中断后修改其对应的寄存器、标志寄存器、寄存器指向的内存数据，在不破坏目标文件的情况下实现对其破解、通杀。

工具有加壳保护，杀软可能会误报工具及补丁文件！由于使用了壳的 SDK，工具所有组件（包括生成的补丁）均**不含联网功能**！所生成的补丁运行时不会修改系统中的任何文件（覆盖补丁文件除外）。工具自身有校验机制，启动时校验模块成功后才会进行加载，但为了安全考虑，请务必从官方下载使用。

### （二）适用范围

Baymax 适用于 Win x86/x64 平台。已适配 WinXP、Win7、Win8、Win8.1、Win10、Win11 等系统，理论上对所有可加载劫持模块的进程进行补丁，支持部分.NET 程序，基本不支持脚本语言程序。

### （三）如何检测

Baymax 旨在快速验证补丁方案，仅限于技术交流，请勿用于侵权

行为或商业用途。

Baymax 的功能模块是 PYG.DLL/PYG64.DLL，功能模块内置简单的文件名称校验，修改后将不执行补丁操作。所以，您可以通过检查进程目录下是否存在该文件，或检查进程是否加载了该模块作为检测依据。

#### （四）免责声明

本工具仅供技术交流，请勿用于商业或非法用途！您创建的补丁文件造成的一切影响和本作者无关！当您使用工具时点击确定免责声明后，代表您已经同意本条款。

## 二．创建补丁项目

#### （一）设置目标进程


补丁项目首先选定一个目标文件，界面中设置的目标程序将作为所生成的补丁程序智能释放劫持文件的参考依据。补丁程序若未找到该文件，则会弹框提示用户选择目标对象。



## （二）添加补丁项并设置补丁进程和模块

添加一个搜索替换或异常中断补丁项，工程中设定的目标进程为各补丁项默认的进程和补丁模块，可自行调整。

Baymax 支持补丁工程中包含多个补丁项，每个补丁项均可设置补丁不同的进程，也可设置补丁所有进程（勾选目标为任意进程），支持对进程的不同 DLL 或其他加载模块（设置目标为其他模块，可在每个补丁项中设置该进程加载的一个模块）进行补丁。



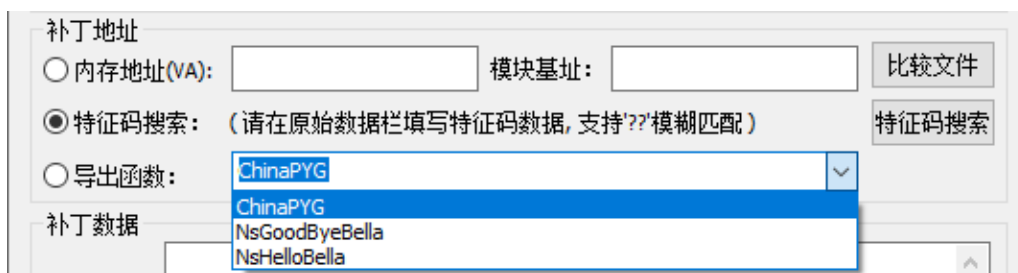
The screenshot shows a configuration window with two rows of input fields. The first row is labeled '文件' (File) and '进程:' (Process:), with a text box containing '\*.exe' and a button with three dots. To the right is a checkbox labeled '目标为任意进程' (Target any process) which is checked. The second row is labeled '模块:' (Module:) and contains a text box with 'Bella64.dll' and a button with three dots. To the right is a checkbox labeled '目标为其他模块' (Target other module) which is also checked.

当您勾选了目标为其他模块时，无需关心该模块何时加载，是否会卸载，或再次加载，若补丁模块非 exe 进程，Baymax 将默认监控模块的加载并尝试补丁。当然您也可以自行处理模块的补丁时机，可在菜单上选择“自定义补丁 DLL 时机”来关闭智能监控功能。

## （三）设置补丁地址

### 1. 偏移补丁

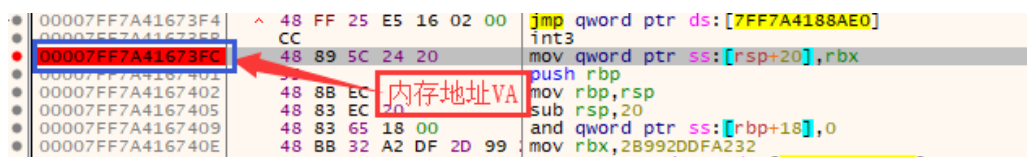
“搜索替换补丁”简称偏移补丁，支持以下 3 种补丁地址的设置方案：



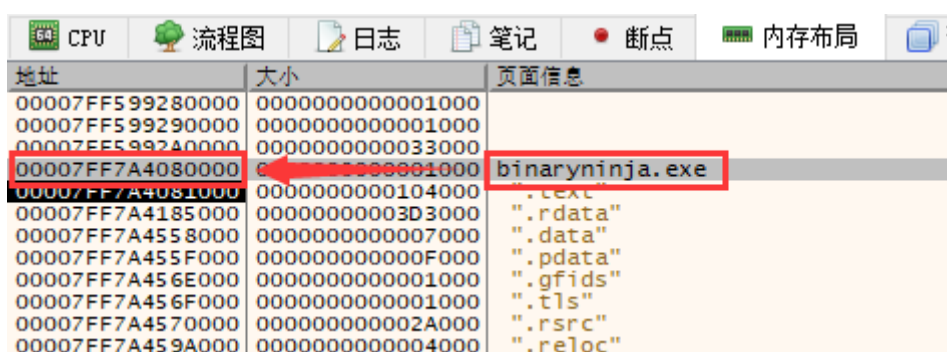
- (1) 输入补丁虚拟地址（内存地址）和模块基地址
- (2) 通过特征码定位补丁地址
- (3) 通过导出函数定位补丁地址

Baymax 支持动态基址（ASLR），通过用户输入的补丁地址和模块地址自动计算 RVA 来进行补丁（用户不需要关心）。

方案 1 中“虚拟地址 (VA)”即我们在调试器中看到的地址，直接输入即可。



“模块基址”需通过调试器查看进程内存，模块的加载地址即模块基地址。



方案 2 通过特征码定位补丁地址，特征码支持 ?? 通配符。使用该方案时，建议使用“特征码搜索”小工具来检查结果是否符合预期。

方案 3 若模块存在导出表，可设置导出函数为补丁地址。

## 2. 中断补丁

“异常中断补丁”简称中断补丁，也支持 3 种补丁地址的设置方案：



The screenshot shows a configuration window for 'Exception Interrupt Patch'. It has three radio buttons for 'Interrupt Address': 'Virtual Address (VA)', 'Feature Code Search', and 'Export Function'. The 'Feature Code Search' option is selected. There are input fields for 'Module Base Address', 'First Byte', 'Offset', and 'Export Function' (set to 'ChinaPYG'). A 'Feature Code Search' button is also present. At the bottom, there are checkboxes for 'Modify Return Value' and 'Stack Adjustment', and a dropdown for 'Return Address Stack Offset' set to 'Save current context and modify on return'.

- (1) 输入补丁虚拟地址和模块基地址
- (2) 通过特征码定位补丁地址
- (3) 通过导出函数定位补丁地址

方案 1 与偏移补丁不同之处在于，若目标模块有壳或待解码，设置 INT3 断点还需要输入补丁地址对应指令的首字节，补丁时先判断内存字节是否等于用户输入的原始数值，来判定该地址是否已解码完毕。若模块无壳或使用硬件断点保持默认值 0 即可。

方案 2 可使用特征码的获取补丁地址，通过搜索结果+偏移值计算出虚拟地址后设置断点（偏移值可为负数）。需要注意的是：Baymax 将会在搜索的所有结果+偏移值设置中断，若搜索结果超过 4 个，使用智能断点或硬件断点可能造成遗漏。在提取特征码后，务必要使用特征码搜索工具检查结果是否符合预期，若在非预期的地址设置了中断，可能会造成程序崩溃。

方案 3 某些壳会检测 API 首字节是否被修改，所以提供了在函数+偏移处下断的功能（偏移值可为负数），这里需要注意的是某些函数在不同平台代码可能不同，若设置了偏移值，在非指令首地址处设置



了中断，将出现未知结果。

#### （四）补丁加壳的程序

Baymax 支持补丁加壳的程序，若目标模块有壳，或补丁地址待解码，破解模块不能在加载后立即执行补丁操作，需等待代码段解码。用户可以用以下几种方案来设定目标进程的解码时机，待解码后方可执行补丁。勾选“补丁数据需程序解码后进行判断”后在下图弹框中进行设置。



##### 1. 使用 HOOK 模式（推荐）

选择合适的 API 通过 HOOK 判断解码时机，从而破解加壳程序是一门艺术，这是经验的积累，新手需要多摸索实践方可掌握。这里可以分享一个 HOOK 方法：壳将数据解码后，执行到补丁地址前，这段时间内执行过的 API 均可作为备选（API 执行次数越少越好）。

该方案补丁加壳的程序是通过 HOOK 目标函数，进程每次调用该函数时都会尝试执行一次补丁方案，直到某次调用时已解码完成并执行补丁。所选函数调用次数越少越好，调用次数越少，对进程的性能损耗越小。

如何选取合适的 API？以下方案可供参考：

1. 壳常用的 API（堆空间申请释放，内存属性修改）、模块入口处调用的 API、模块导入表中的 API、行为操作相关的 API（如窗口创建销毁），调试信息输出等 API，多试几次总会找到合适的。

2. 通过调试精准定位，找到程序解码后 OEP 附近调用的 API，或者从补丁地址向上回溯，尝试程序调用过的 API。

若选择的 API 导致程序无法启动或启动后未补丁，请更换其他 API 进行测试。找到可用的 API 之后，我们再挑选一个调用次数尽量少的 API，这样可以降低进程启动的性能损失。如何确定调用次数的多少呢？可以在调试器中判定设置中断次数，也可通过 Baymax 调试补丁的日志统计输出 HOOK 函数执行的情况。

采用该方案后，补丁条目不会在 PYG 模块加载时进行补丁，而是进程每次执行到所选 API，都会判定一次是否解码并尝试补丁，直到对应的补丁条目全部成功设置或补丁，再次执行到所选 API 时才不会继续补丁。

如果需要增加“选择模块”列表中系统 DLL，可在 Baymax 的 ini 配置项[HOOKDLL]中自行添加系统模块。

## 2. 设置中断模式

若目标进程没有检测硬件断点，例如未加壳或压缩壳，我们可设置中断模式，选择模块并输入偏移地址 RVA（内存地址-模块基址）即可。

采用该方案后，补丁条目不会在 PYG 模块加载时进行补丁，而是

当进程执行到该地址触发硬件断点时，才开始执行补丁操作。

### 3. 设置延时模式

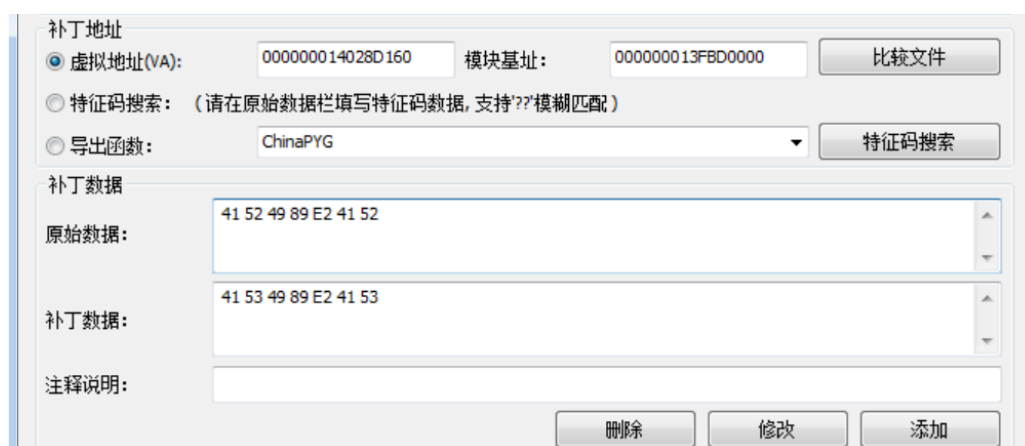
程序从启动到运行至补丁地址必有时间间隔，有时我们不想关心程序壳的解码过程，可设置一个等待时间再进行补丁。此方案是懒人法，可能不同的系统环境会造成未知结果，不通用。

采用该方案后，补丁条目不会在 PYG 模块加载时进行补丁，破解模块在加载时将创建等待线程，新线程当满足用户设定的等待时间后，才会执行补丁操作。

注：输入的毫秒数为十进制（软件设置界面中唯一一处输入 10 进制的地方，1 秒=1000 毫秒）。

## 三．搜索替换补丁

### （一）偏移补丁



添加一条偏移补丁条目：

首先输入补丁地址：可选择方案一中输入内存地址和模块基址，或选择方案三中的导出函数。

然后输入补丁地址对应的原始数据和补丁数据（偏移补丁不支持通配符），点击“添加”按钮完成操作。

若补丁数据较多，可使用“比较文件”功能，对比原文件和修改后的文件将补丁数据一次性添加。

## （二）特征码替换补丁

00007FF7A4167409	48:8365 18 00	and qword ptr ss:[rbp+18],0
00007FF7A416740E	48:BB 32A2DF2D992B0000	mov rbx,2B992DDFA232
00007FF7A4167418	48:8B05 D1163F00	mov rax,qword ptr ds:[7FF7A4558AF0]
00007FF7A416741F	48:38C3	cmp rax,rbx
00007FF7A4167422	75 6F	jne binaryninja.7FF7A4167493
00007FF7A4167424	48:8D4D 18	lea rcx,qword ptr ss:[rbp+18]
00007FF7A4167428	FF15 F2DB0100	call qword ptr ds:[&GetSystemTimeAsFileTime]
00007FF7A416742E	48:8845 18	mov rax,qword ptr ss:[rbp+18]
00007FF7A4167432	48:8945 10	mov qword ptr ss:[rbp+10],rax
00007FF7A4167436	FF15 ECDB0100	call qword ptr ds:[&GetCurrentThreadId]

添加一条特征码搜索补丁条目：

为了让补丁支持后续版本，我们经常从补丁地址提取出特征码，制作成通杀补丁。我们用“特征码搜索”工具（或 baymax plug-in for x64dbg）来检索我们提取的数据，以确保结果符合预期。

特征码的提取，需要避开指令中的相对偏移地址，因为后续版本相对偏移地址可能发生变化。如上述方框中的指令，就有两条汇编代码中包含了相对偏移地址（红色箭头标准）。

00007FF7A416740E	48:8B 32A2DF2D992B0000	mov rbx,2B992DDFA232
00007FF7A4167418	48:8B05 D1163F00	mov rax,qword ptr ds:[7FF7A4558AF0]
00007FF7A416741F	48:8B05 CA163F00	mov rax,qword ptr ds:[7FF7A4558AF0]
00007FF7A4167426	4D:18FF	sbb r15b,r15b
00007FF7A4167429	15 F2DB0100	adc eax,1D8F2
00007FF7A416742E	48:8845 18	mov rax,qword ptr ss:[rbp+18]
00007FF7A4167432	48:8945 10	mov qword ptr ss:[rbp+10],rax
00007FF7A4167436	FF15 ECDB0100	call qword ptr ds:[&GetCurrentThreadId]
00007FF7A416743C	8BC0	mov eax,eax

如上图，由于指令中的相对偏移在后续版本中极有可能发生变更，所以我们用 ?? 通配符进行替换。可以对比下图中提取的特征码数据。

补丁地址  
☐ 内存地址(VA):  模块基址:    
☒ 特征码搜索: (请在原始数据栏填写特征码数据, 支持"?"模糊匹配)   
☐ 导出函数:

补丁数据  
 原始数据: 48 8B 05 ?? ?? ?? ?? 48 3B C3 75 6F 48 8D 4D 18 FF 15 ?? ?? ?? ?? 48 8B 45 18  
 补丁数据: 48 8B 05 ?? ?? ?? ?? 48 3B C3 75 00 48 8D 4D 18 FF 15 ?? ?? ?? ?? 48 8B 45 18  
 注释说明:

选择方案 2 特征码搜索，输入原始数据和补丁数据后，点击“添加”按钮完成操作。输入栏支持特征码数据中包含回车空格，但有效数据的长度必须相同。

推荐使用 baymax plug-in for x64dbg 来获取特征码数据：

特征码搜索

模块列表: ntdll.dll | ModuleBase:000077480000 SizeOfImage:1A9000

特征码: E9 ?? ?? ?? ?? 8B 0D ?? ?? ?? ?? F6 C1 03 74 31

☐ 搜索字符串

序号	内存地址(VA)	相对偏移(RVA)	文件偏移(FOA)	区段名称	内存属性
1	0000774E224D	0006224D	0006164D	".text"	ER---
2	0000774E2F20	00062F20	00062320	".text"	ER---
3	00007751B6B8	0009B6B8	0009AAB8	".text"	ER---

共找到 3 处, 耗时<1ms 模块列表 ☐ 搜索列表所有项

## 四. 异常处理补丁

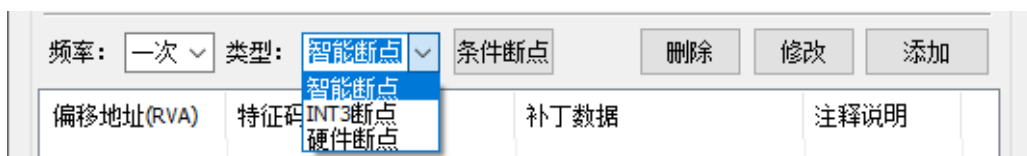
### (一) 设置中断地址

我们使用调试器分析程序，先对模块设置断点，运行中断之后再  
进行调试分析，异常中断补丁的设计思路亦如此，首先要设置中断地  
址，中断后方可进行补丁操作。

每个中断地址都可以设置频率为中断一次或每次都中断，每个中  
断地址都支持设置多个补丁条目，每个补丁条目支持设置不同的条件  
断点。

### (二) 中断类型

中断类型目前有 3 种：智能断点、INT3 断点、硬件断点。



1. 硬件断点：硬件断点有线程相关性，每个线程可以同时设置  
4 个硬件断点，默认将设置于破解模块加载时的线程上（一般为主线  
程），若使用了 HOOK API，将设置到所执行该 API 的对应线程。目前  
未提供对所有线程设置硬件断点的功能。

设置硬件断点只修改线程上下文中 DrX 寄存器，不会修改进程内  
存字节，不会触发进程的内存校验。

2. 智能断点：由于每个线程同时最多只能设置 4 个硬件断点，  
所以当用户同时设定的硬件断点超过 4 个时，将设置失败。此时用户

可以选择智能断点，智能断点本质上等于硬件断点，每次触发硬件断点，执行结束时若有空闲的 DrX，会检测队列中是否还有未设置的断点并依次进行设置，队列中的先后顺序和用户在界面中添加顺序一致，若设置合理，可以间接的设置使用多个硬件断点。

3. INT3 断点：INT3 不同于硬件断点，无线程相关性，设置后进程所有线程执行到该地址都会触发中断。若设置硬件设置无效，可能是因为该线程未设置上断点，可修改为 INT3 断点进行测试。

使用 INT3 断点将修改补丁地址指向的内存为 0xCC，执行结束后将恢复为原始指令。INT3 断点模式未做多线程兼容，若多线程同时执行到补丁地址，则可能出现不可预知的结果。

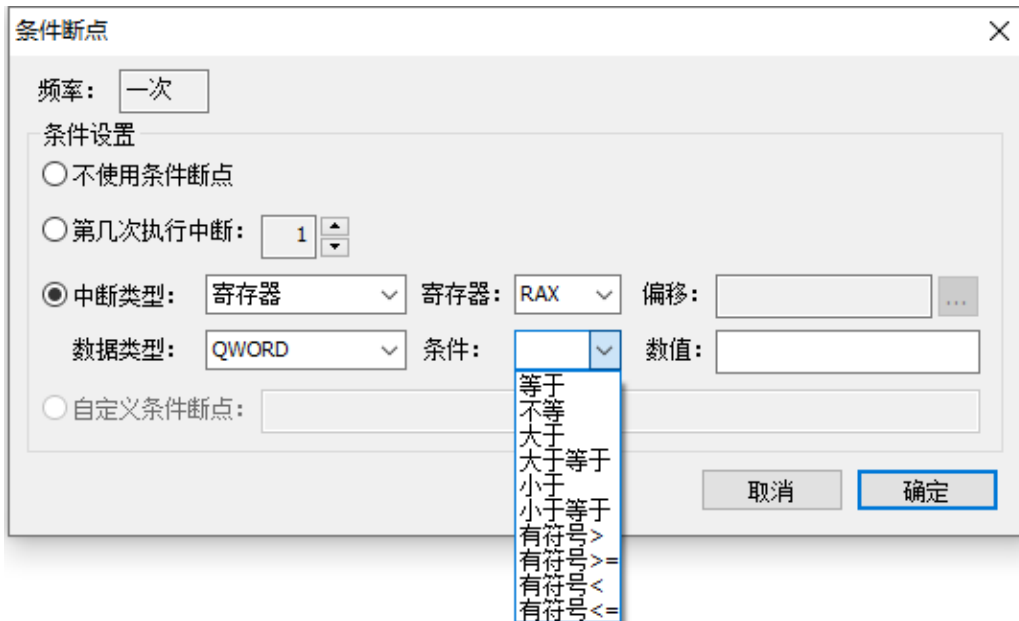
### （三）条件断点

#### 1. 判断第几次中断



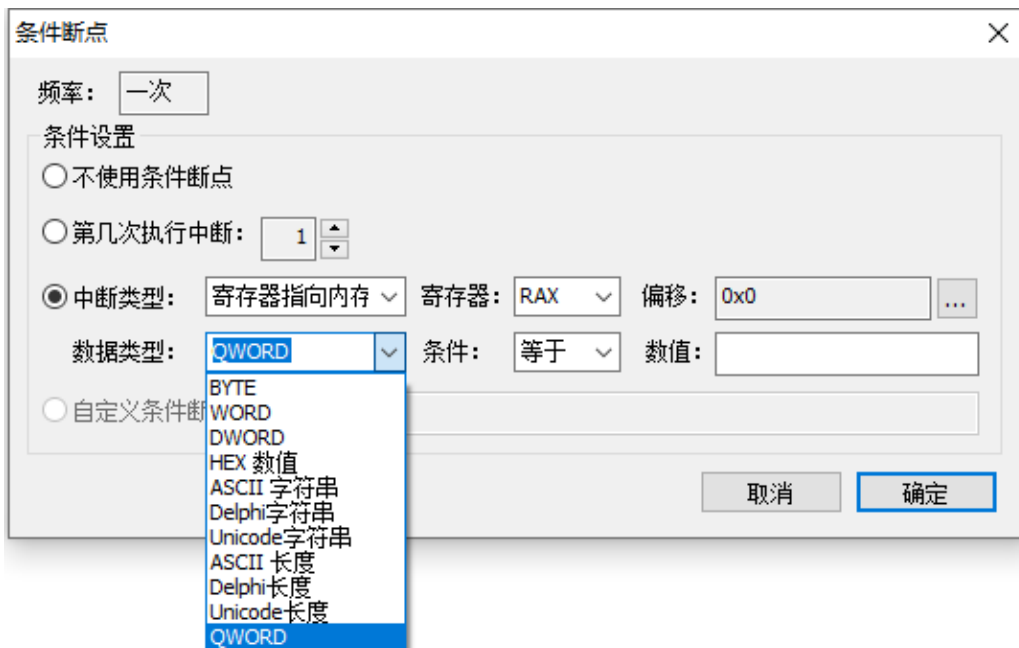
条件断点可以设置补丁地址在第几次中断后执行补丁方案。

#### 2. 判断寄存器数值



条件断点可以比较中断后寄存器的数值，来作为是否执行补丁的依据。

### 3. 判断寄存器指向的内存数值



条件断点可通过获取寄存器指向的内存数据，来作为是否执行补丁的依据。

### 4. 通过寄存器偏移定位内存地址

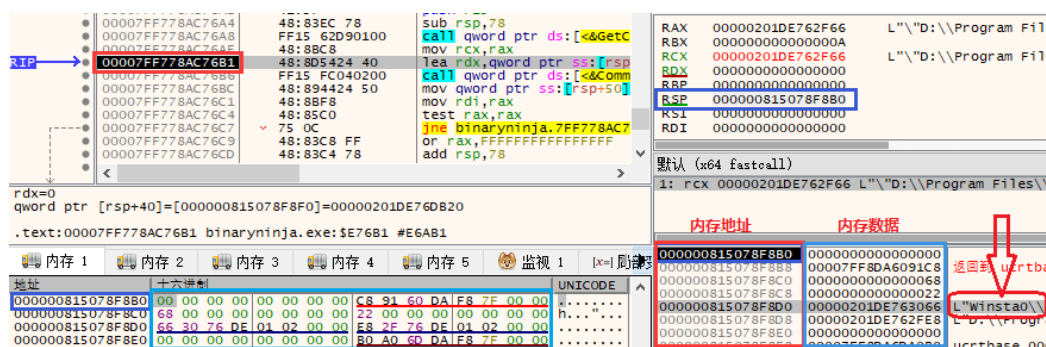




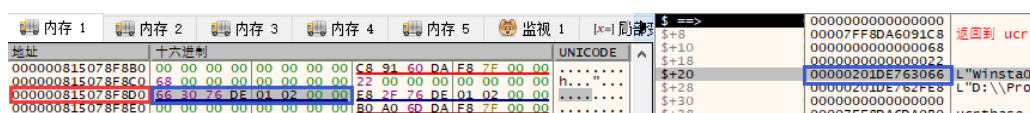
条件断点和后续的补丁设置都需要掌握如何设置“寄存器指向的内存”，这里统一进行讲解。

通过寄存器+偏移用来定位数据的内存地址，“补丁内存地址”栏输入的一定指向需补丁数据的内存地址，而非内存数据。若内存数据无法通过中断地址对应的寄存器+偏移来定位，则无法使用本方案取值或赋值。

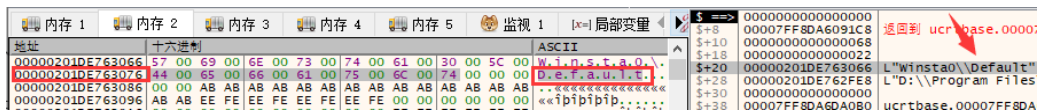
中断地址处，我们首先选定一个寄存器，界面默认显示的偏移值为 0x0（这里输入的所有数据均为十六进制）。如上图输入栏默认数据为：RAX+0，代表取寄存器 RAX+偏移为 0 的数值作为内存地址。



上图中 RIP: 00007FF778AC76B1, RSP: 000000815078F8B0, 我们如何通过寄存器来定位图中字符串 L "winsta0"的地址呢？



我们观察到 RSP 的偏移 0x20 处内存地址为 000000815078F8D0 (RSP+0x20),该地址指向的内存数据 QWORD 值为 00000201DE763066,正是字符串的地址。所以我们先设置 RSP 偏移为 20,即 RSP+20,再取其值即可,即输入[RSP+20]。

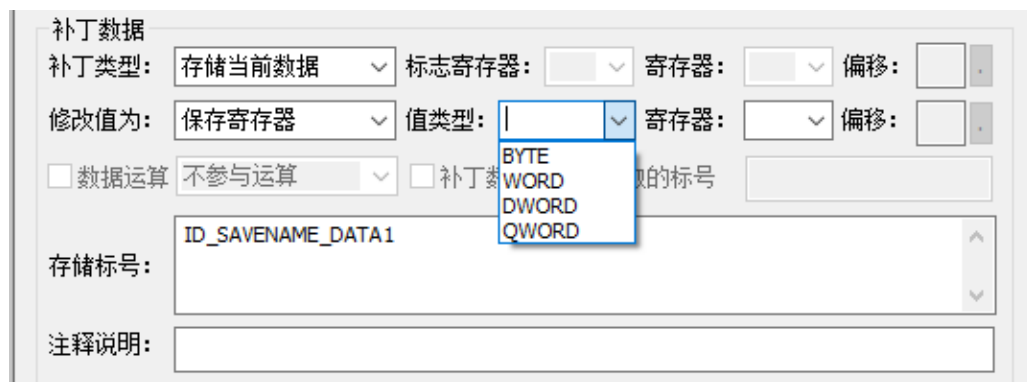


那么如何定位字符串 L "Default"的地址呢? 首先定位到字符串地址后再+0x10 的偏移即可,即输入[RSP+20]+10。

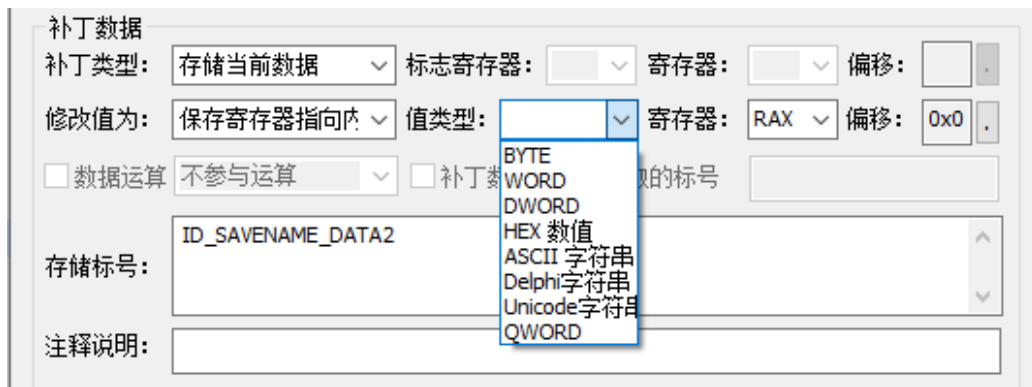
Baymax 支持对多层偏移寻址,这样通过类对象地址寻址其内部的成员将变得更加灵活便捷。

#### (四) 存储数据

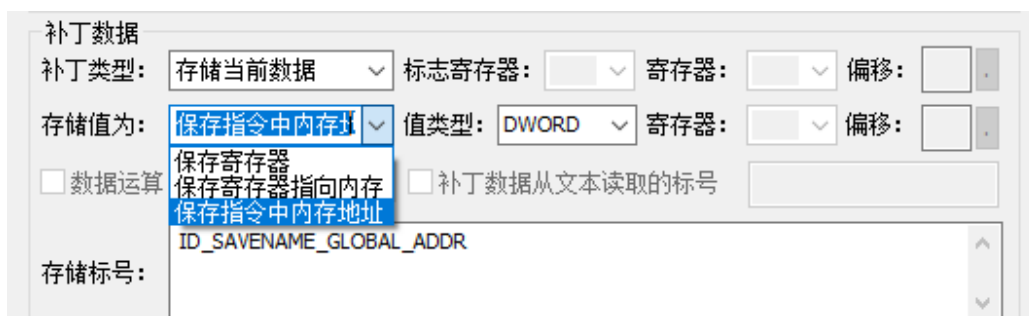
触发中断时,可以存储中断地址对应的寄存器值或寄存器指向的内存数据。



选择存储值类型和对应的寄存器,并在“存储标号”栏输入我们对该存储数据的命名的唯一名称(全局变量名),方便我们后续使用时引用。注意存储时设定的数据类型,与后续引用时必须保持类型一致。



也可以存储通过寄存器+偏移所指向内存的数据，可保存更多类型的数据。当值类型为“HEX 数值”时，“存储标号”栏数据为“ID\_SAVENAME\_,0”，ID\_SAVENAME\_为该存储数据的名称，逗号之后为 HEX 数据的长度。



存储当前数据还支持从中断地址处的汇编指令中提取形如[XXXX]中的数值，即存储指令中的全局变量的地址。

7738E9FB | 890D 74E73F77 | mov dword ptr ds:[773FE774],ecx

如中断地址为 7738E9FB,存储值选择“保存指令中的内存地址”，设置存储标号，中断后将存储 DWORD 数值 773FE774。

保存该地址后，我们可在补丁类型为“修改存数指向的内存”中使用该存储标号，并修改该存储标号指向的内存数据，即可修改指令中全局变量的数值。

## （五）修改寄存器

触发中断时，可以修改中断地址对应的寄存器值。

补丁数据

补丁类型: 修改寄存器 标志寄存器: 寄存器: RAX 偏移:

修改值为: 自定义立即数 值类型: QWORD 寄存器: 偏移:

☐ 数据运算 ☐ 补丁数据从文本读取的标号

补丁数据:

注释说明:

补丁数据可以为自定义数据，在补丁数据栏输入即可，注意数值不可超过所选的值类型；也可以选择其他寄存器，例如中断时寄存器分别指向真假码，可将存储假码的寄存器数值替换为存储真码的寄存器值；也可以替换为寄存器指向内存的数据。

补丁数据

补丁类型: 修改寄存器 标志寄存器: 寄存器: RAX 偏移:

修改值为: 使用存储数据 值类型: QWORD 寄存器: 偏移:

☐ 数据运算 不参与运算 ☐ 补丁数据从文本读取的标号

存储标号: ID\_SAVENAME\_DATA1

注释说明:

还可以使用我们曾存储过的数据，使用存储数据必须保证值类型一致。存储标号栏输入我们曾保存数据的名称。

## （六）修改标志寄存器

触发中断时，可以修改中断地址对应的标志寄存器。

修改指令跳转，可以通过修改标志寄存器来改变执行流程，同一个中断地址可以添加多个补丁条目以实现修改多个标志寄存器，在“寄存器值”中填写修改值 0 或 1。

## (七) 修改寄存器指向的内存

触发中断时，可以修改中断地址寄存器指向的内存数值。

修改值支持自定义立即数（例如替换 RSA 的公钥或解码后的数据），支持使用当前寄存器值，支持使用寄存器指向的内存（例如两个寄存器指向的内存数据为比较数值，可在比较前进行替换），也可使用存储数据。

上文“条件断点”章节中详细讲解过通过寄存器定位内存的方法，原则为输入的数据一定为内存数据的地址。例如我们在寄存器中看到

的字符串，其地址就等于寄存器值：

004BF7DC - B8 4D5A0000 mov eax, 5A4D  
004BF7E1 - 66:3905 0000 cmp word ptr [eax], 0005  
004BF7E8 - 75 38 jnz short 004BF7F0  
004BF7EA - A1 3C004000 mov eax, dword ptr [esp+10B]  
004BF7EF - 81B8 00004000 cmp dword ptr [eax], 00004000  
004BF7F9 - 75 27 jnz short 004BF7F0  
004BF7FB - B9 0B010000 mov ecx, 10B  
ds:[0040003C]=000000F8  
eax=0005A4D

寄存器 (FPU)  
EAX 0005A4D  
ECX 00000081  
EDX 0019FF08 ASCII "Ls "  
EBX 00253000  
ESP 0019FEF8 ASCII "Baymax Patch to0Ls "  
EBP 0019FF70  
ESI 00000000  
EDI FFFFFFFE  
EIP 004BF7EA BestCryp.004BF7EA

0019FEF8 42 61 79 6D 61 78 20 50 61 74 63 68 20 74 6F 4F Baymax Patch to0  
0019FF08 4C 73 20 00 5C 28 93 00 3C 28 93 00 D0 27 93 00 Ls .\{??(???

补丁内存地址输入：ESP 或 ESP+0 均可。

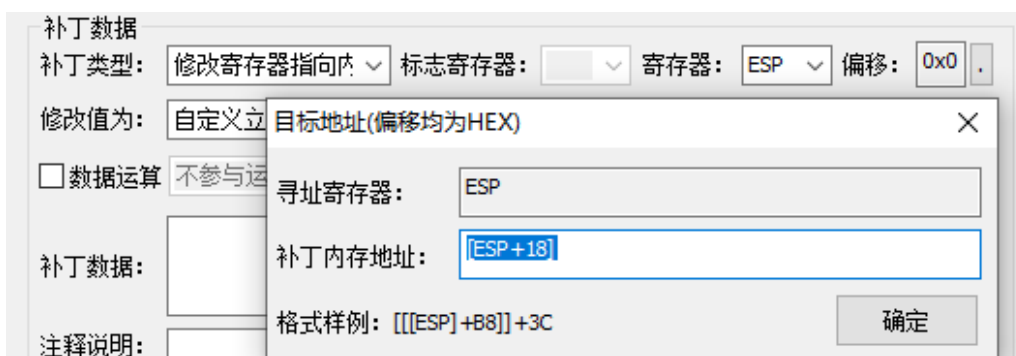
补丁数据  
补丁类型：修改寄存器指向内存 标志寄存器： 寄存器： ESP 偏移： 0x0  
修改值为：自定义立即数 目标地址(偏移均为HEX)  
☐ 数据运算 不参与运算  
补丁数据：  
注释说明：  
寻址寄存器： ESP 输入要修改数据的内存地址  
补丁内存地址： ESP+0  
格式样例： [[[ESP]+B8]]+3C 确定

而在堆栈中看到的字符串，其地址就需要先通过 ESP/RSP 加上偏移定位到字符串行，再取该地址的数据，就可以获取字符串的地址了。

EAX 00000000  
ECX 00000081  
EDX 0019FF08  
EBX 003FA000  
ESP 0019FEF8  
EBP 0019FF70

\$ ==> 0FAB0BC0  
\$+4 004BF938 BestCryp.<模块入口点>  
\$+8 004BF938 BestCryp.<模块入口点>  
\$+C 003FA000  
\$+10 00000044  
\$+14 0074285C  
\$+18 0074283C UNICODE "WinSta0\Default"  
\$+1C 007427D0 UNICODE "D:\Program Files (x86)\Jet:  
\$+20 00190000

如上图，ESP+18=0019FF10，该地址指向的内存数据为 0074283C，0074283C 才是字符串“WinSta0...”的地址。



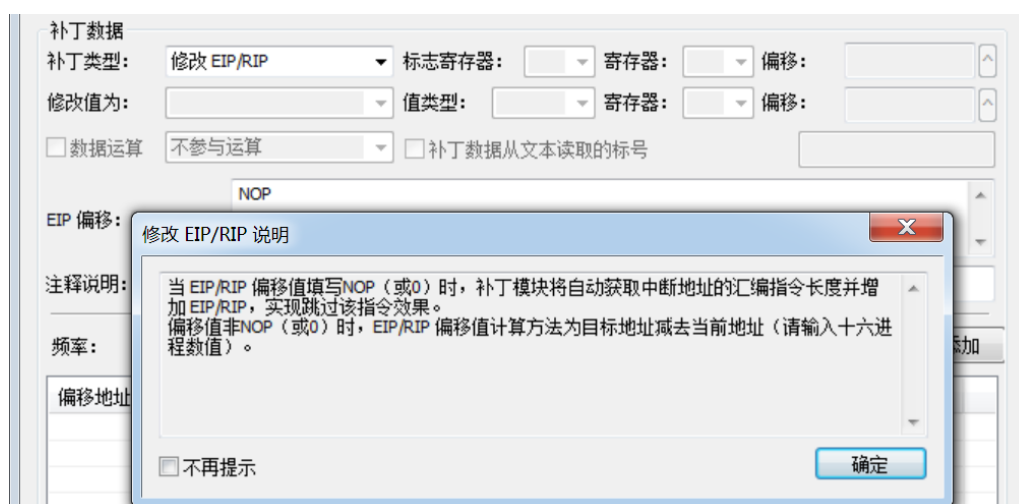
此时，补丁内存地址应输入：`[ESP+18]`。

输入格式支持多层寻址，格式为：`[[EXX+X]+Y]+Z`，其中 EXX 为寄存器，XYZ 为偏移值，`[]` 代表取该内存指向的数据，32 位进程取值为 DWORD，64 位进程为 QWORD。

小总结：取寄存器指向字符串的地址：`EXX+0/RXX+0`。取栈中字符串的地址：`[ESP+N]/[RSP+N]`。

## （八）修改 EIP/RIP

触发中断时，可以修改 EIP/RIP 值。



偏移值栏默认的为 NOP，其实这里并非修改指令为 NOP，而是补丁模块将自动获取中断地址的汇编指令长度并增加 EIP/RIP，实现跳过该指令效果。也可以自定义偏移值，偏移值非 NOP(或 0)时，EIP/RIP

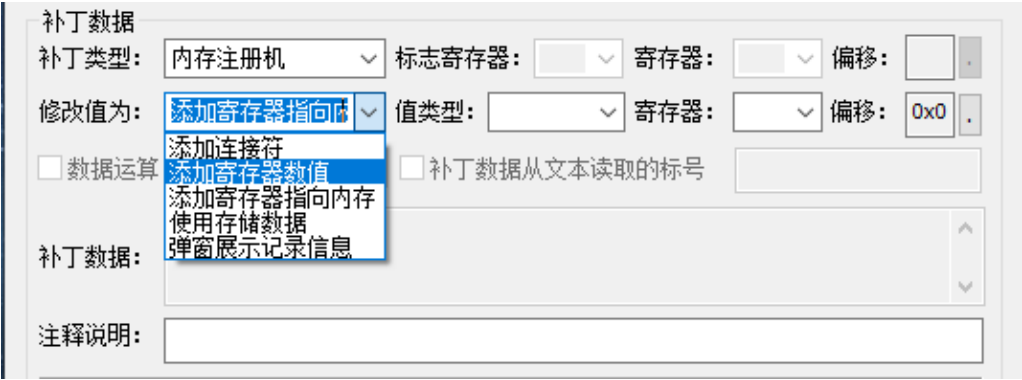


偏移值计算方法为目标地址减去当前补丁地址（请输入十六进制数值），支持正负数。

### （九）制作内存注册机

Baymax 提供了一种弹框显示提取信息的功能，可先采集很多信息，最后一次性进行展示。该功能也可用来做内存注册机之用。

实现分两步走，第一步按顺序依次添加数据，支持从寄存器或内存中读取信息，支持添加连接字符，也可以添加存储数据。



第二步，展示截取信息，当添加完所有待展示信息后，可以在任意地址设置中断，然后添加一项“弹窗展示记录信息”，则将在执行到该地址触发中断时候，创建新线程将之前添加的所有数据按顺序拼接并进行弹窗展示。另外当执行到弹框地址时，log 文件中将记录该地址所有寄存器的信息。



补丁数据

补丁类型: 内存注册机 标志寄存器: 寄存器: 偏移:

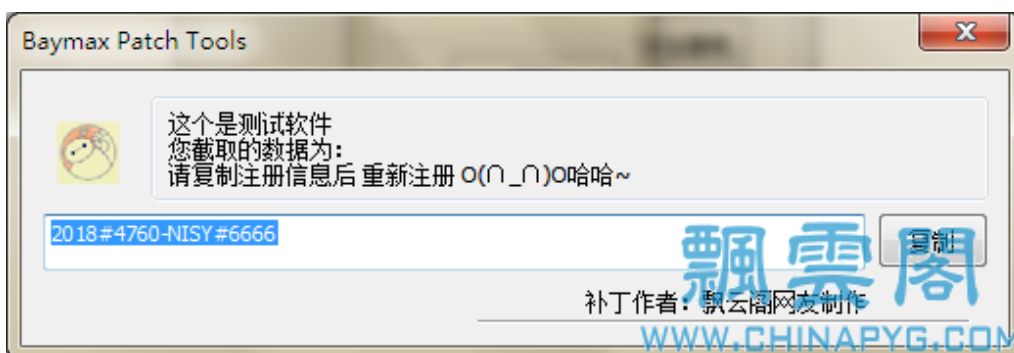
修改值为: 弹窗展示记录信息 值类型: 寄存器: 偏移:

☐ 数据运算 不参与运算 ☐ 补丁数据从文本读取的标号

补丁说明: 您截取的数据为:

注释说明:

补丁说明栏可以自定义弹框信息，展示界面如下图：



## （十）修改存储地址指向的内存

若存储的数据为地址，我们可在任意地址设置中断，修改该存储地址指向的内存。

补丁数据

补丁类型: 修改存数指向的内 标志寄存器: 寄存器: RXX 偏移: 0x0

修改值为: 寄存器指向的内存 值类型: HEX 数值 寄存器: 偏移: 0x0

☐ 数据运算 不参与运算 ☐ 补丁数据从文本读取的标号

标号数据: ID\_SAVENAME\_DATA1,0x1

注释说明:

修改值可以为自定义立即数，中断地址的寄存器或寄存器指向的内存数据。输入栏格式为“ID\_SAVENAME\_,0”，ID\_SAVENAME\_为我们之前保存的 DWORD（32 位进程）或 QWORD（64 位进程）名称，逗号之后

为我们输入自定义数据，其中修改值为“寄存器指向的内存”时，若选择“HEX 数值”，逗号后请输入 HEX 数据的长度。

## （十一）添加反调试检测

Baymax 支持对中断地址设置反调试检测，当调试器动态分析时，若断点处停留时间超过用户预设值，则假设程序可能正在被调试分析，程序将触发崩溃。

补丁数据

补丁类型: **添加反调试检测** 标志寄存器: 寄存器: 偏移:

补丁数据: 值类型: 寄存器: 偏移:

☐ 数据运算 不参与运算 ☐ 补丁数据从文本读取的标号

中断间隔: 200 **单位为毫秒，格式为十进制。  
默认值200，最小值50，最大值2000**

用户预设值单位为毫秒，输入栏为十进制数值，默认 100，最小值 20（建议不小于 50 毫秒），最大值 2000（2 秒）。

## （十二）修改函数的返回值

有时我们需要修改函数的参数或返回值，Baymax 没有提供函数 HOOK 功能，但通过异常中断来模拟实现了该功能。

中断补丁项中所有的补丁条目，均可使用“修改返回值”功能。补丁地址中断后，读取函数返回地址并设置中断，函数返回后再次触发中断并执行补丁操作。

函数返回过程可分解为两步：首先 Set New EIP/RIP，然后 ESP/RSP + N 恢复栈平衡。模拟函数返回亦如此，所以以下三种修改

返回值的模式，同样需要关注这两组信息：从栈中获取函数返回地址和栈平衡调整值。

### 1. 函数调用约定

我们需要了解一下栈平衡知识，这里对常见的函数调用约定做一下简单介绍：

1. `__cdecl` 是 C/C++ 默认的函数调用协议。所有参数从右往左依次入栈，函数返回一般为 `retn`，栈平衡由调用方来平衡，调用方在函数返回后会执行 `add esp, n` 来平衡堆栈，`n` 为参数压栈所占的内存大小。

005CBEA0	. 8B56 20	mov	edx, dword ptr [esi+20]	
005CBEA3	. 52	push	edx	压入参数1 ESP-4
005CBEA4	. 8B4E 1C	mov	ecx, dword ptr [esi+1C]	
005CBEA7	. 51	push	ecx	压入参数2 ESP-8
005CBEA8	. E8 DBD3FFFF	call	005C9288	
005CBEAD	. 83C4 08	add	esp, 8	ESP+8 保证栈平衡

函数实现见下图，`retn` 代表函数无参数或使用 `cdecl` 调用协议。

005C9288	55	push	ebp	
005C9289	. 8BEC	mov	ebp, esp	
005C928B	. 8B45 08	mov	eax, dword ptr [ebp+8]	函数实现
005C928E	. 8B55 0C	mov	edx, dword ptr [ebp+C]	
005C9291	. A3 284A6600	mov	dword ptr [664A28], eax	retn 可说明无参数或
005C9296	. 8915 2C4A6600	mov	dword ptr [664A2C], edx	使用_cdecl调用模式。
005C929C	. 5D	pop	ebp	
005C929D	. C3	retn		

2. `__stdcall` 是 Windows API 默认的的函数调用协议。所有参数从右往左依次入栈，栈平衡由函数自身实现平衡，函数返回一般为 `retn n`，调用方无需关注栈平衡。

00401017	. C1E0 02	shl	eax, 2	
0040101A	. A3 63675E00	mov	dword ptr [5E6763], eax	
0040101F	. 52	push	edx	
00401020	. 6A 00	push	0	
00401022	. E8 6B401E00	call	< jmp.&KERNEL32.GetModuleHandleA>	pModule = NULL
00401027	. 8BD0	mov	edx, eax	stdcall 模式，由函数负责栈平衡
00401029	. E8 A2FD1B00	call	005C0DD0	
005E0922	= < jmp.&KERNEL32.GetModuleHandleA>			
005E0922	00 20 08 BA 4E 00 00 20 24 21 51 00 00 00 C8 07	. 00000000 \$!Q...?		ESP 0019FF6C
005E0929	5C 00 00 20 D4 07 5C 00 00 1C 68 10 5D 00 00 20	. \..?..\h...?		EBP 0019FF80
005E092E	74 0E 5C 00 00 20 A0 0F 5C 00 00 00 94 20 5C 00	. t...?...\..?		ESI 00401000 屏录专家-<模块入口点>
				EDI 00401000 屏录专家-<模块入口点>
				EIP 00401022 屏录专家-0040
				C 0 ES 002B 32位 0(FFFFFFFF)
				P 1 CS 0023 32位 0(FFFFFFFF)
				A 0 SS 002B 32位 0(FFFFFFFF)
				Z 1 DS 002B 32位 0(FFFFFFFF)
				pModule = NULL
				屏录专家-<模块入口点>
				返回到 KERNEL32.76B76359

调用系统函数 `GetModuleHandleA` 时，将参数压入堆栈，执行 `CALL`

指令时，栈顶为 0019FF6C。

7620B31A	23DE	and	ebx, esi	
7620B31C	8D45 F4	lea	eax, dword ptr [ebp-C]	
7620B31F	50	push	eax	
7620B320	FF15 40802C76	call	dword ptr [<ntdll.RtlFreeUnicodeString	
7620B326	8BC3	mov	eax, ebx	
7620B328	5B	pop	ebx	
7620B329	5E	pop	esi	
7620B32A	C9	leave		
7620B32B	C2 0400	retn	4	stdcall模式 函数负责平衡栈

系统函数负责栈平衡，该函数执行 ret 4 返回。

0040101A	A3 63675E00	mov	dword ptr [5E6763], eax	[pModule = NULL stdcall 模式, 由函数负责栈平衡	EAX 00000000	ESP 0019FF70	屏录专家.<模块入口点>
0040101F	52	push	edx		ESI 00401000	屏录专家.<模块入口点>	
00401020	6A 00	push	0		EDI 00401000	屏录专家.<模块入口点>	
00401022	E8 6B401E00	call	<jmp.&KERNEL32.GetModuleHandleA		EIP 00401027	屏录专家.00401010	
00401027	8BD0	mov	edx, eax		C 0 ES 0028 32 00 0(FFFFFFFF)		
00401029	E8 A2FD1B00	call	005C00D0	P 1 CS 0023 32 00 0(FFFFFFFF)	A 0 SS 002B 32 00 0(FFFFFFFF)	Z 1 DS 0028 32 00 0(FFFFFFFF)	
0040102E	5A	pop	edx				
eax=00400000 (屏录专家.00400000), ASCII "MZP"							
edx=00401000 (屏录专家.<模块入口点>)							
005E6000	00 20 08 BA 4E 00 00 20 24 21 51 00 00 00 C8 07	; 调整 \$t0...		0019FF70	00401000	屏录专家.<模块入口点>	
005E6010	5C 00 00 20 D4 07 5C 00 00 1C 68 10 50 00 00 20	; ?\?.\mhl...		0019FF74	76B76359	返回到 KERNEL32.76B76359	
005E6020	74 0E 5C 00 00 20 A0 0F 5C 00 00 94 20 5C 00	; ?\?.\?.\?		0019FF78	00389000		
005E6030	00 00 28 23 5C 00 00 05 CC 49 5C 00 00 04 0C 37	; ?\?.\?.\?.\?		0019FF7C	76B76340	KERNEL32.BaseThreadInitThunk	

函数返回后，此时栈顶地址为 0019FF70。因为系统函数负责平衡堆栈，所以若我们在函数中某地址直接返回，栈顶也需要保持和原函数返回时一致，栈平衡的调整需要自己设定。

3. `__fastcall` 为 x64 进程的函数调用协议，参数从右向左顺序传递。参数通过 RCX、RDX、R8、R9 寄存器传递，参数超过 4 个时也会在堆栈中预留前 4 个的栈空间，并将后续参数写入对应的栈偏移。

## 2. 三种修改函数返回值模式

了解这些知识后，我们继续看三种修改返回值的设置方案。

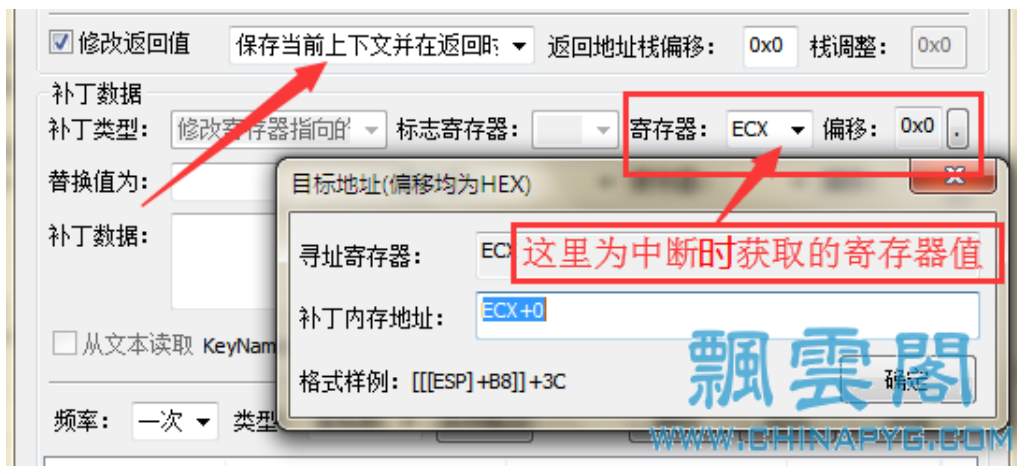
<input checked="" type="checkbox"/> 修改返回值	保存当前上下文并在返回时修改	返回地址栈偏移: 0x0	栈调整: 0x0
补丁数据	保存当前上下文并在返回时修改		
补丁类型: 修改函数直接返回后修改上下文	函数返回时修改上下文	寄存器:	偏移: 0x0

### 1. 保存当前上下文并在返回时修改。

某些函数使用寄存器传参，我们需要修改函数返回时参数接收的数据，但在函数返回处，却无从找寻所传入的参数值。此时需要先保存寄存器数值，当函数返回后再修改参数指向的内存数据。

该需求通过“存储数据”和“修改存储地址指向的内存”也可实现，但不够方便。

采用本方案，用户可在函数内任一可获取参数的地址设置中断，同时设置补丁地址中断时“返回地址栈偏移”，即中断后堆栈中函数返回地址相对于 ESP/RSP 的偏移值。由于该情况传入参数一定为内存地址，所以补丁数据栏中的补丁类型，只能选择“修改寄存器指向的内存”。中断时将设置的寄存器+偏移值将进行保存，函数返回后再修改保存地址指向的内存数据。



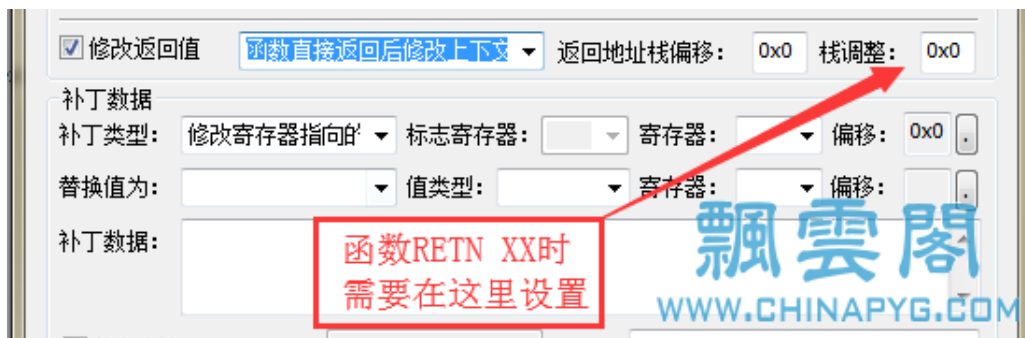
## 2. 函数返回时修改上下文

采用本方案时，也需要设置“返回地址栈偏移”。我们设置的补丁数据并非在补丁中断时执行，而是在中断后从栈中获取函数返回地址并设置断点，函数返回处中断后才开始读取寄存器上下文并执行补丁数据。

## 3. 函数直接返回后修改上下文

有些函数我们不需要其执行，只需要保证函数返回值即可，此时可选择本方案，除了需设置“返回地址栈偏移”，还需要修正“栈调

整”值，以保证函数直接返回后堆栈平衡。该操作可以在函数内任一指令处设置，栈调整请参考函数正常返回时栈顶地址。



### 3. 更好的理解任意指令均位于函数体内

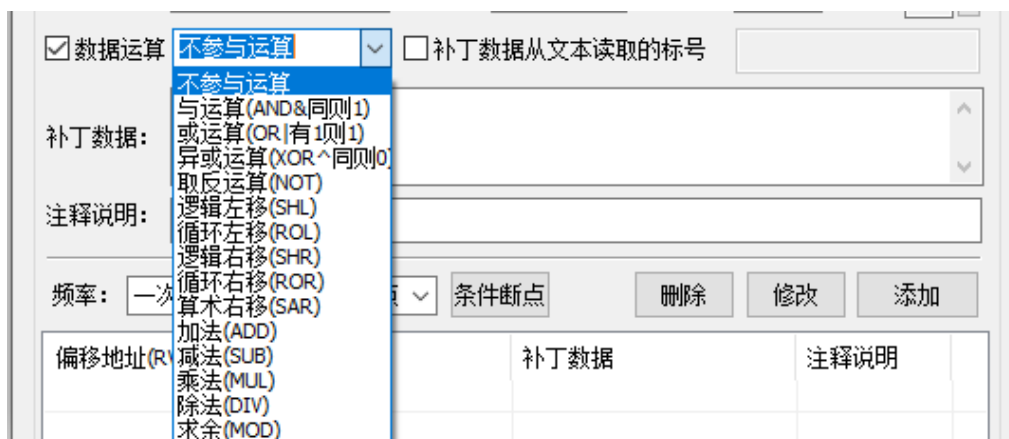
任意一条指令，都处于某个函数中，（基本）都存在函数返回。并非只有 API 函数才可以设置返回，补丁地址均支持修改函数返回值。

### 4. 任意栈地址返回后再进行补丁

修改函数的返回值，并非只能在本函数返回后进行修改，只需在使用前修改即可，所以堆栈中的一些函数返回地址都可以加以利用，补丁工具不关心函数何时返回，只关心从堆栈中获取的返回地址。

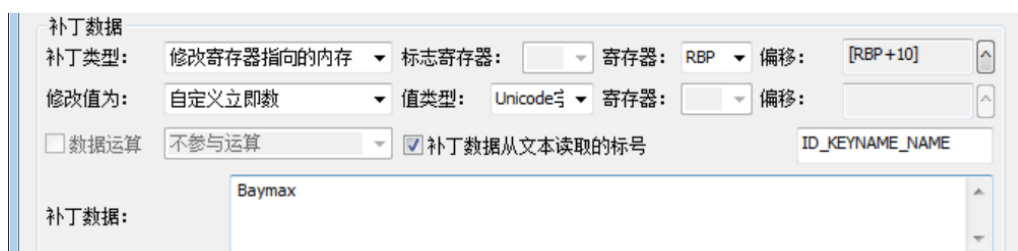
## （十三）对数据进行运算

某些情况，我们需要对数据进行运算操作，当修改值为“自定义立即数”时可勾选“数据运算”选项，补丁数据栏中输入的内容将作为参与运算的第二项数值。选择“取反运算（NOT）”，将忽略输入数据。



#### (十四) 从 INI 文件中读取补丁信息

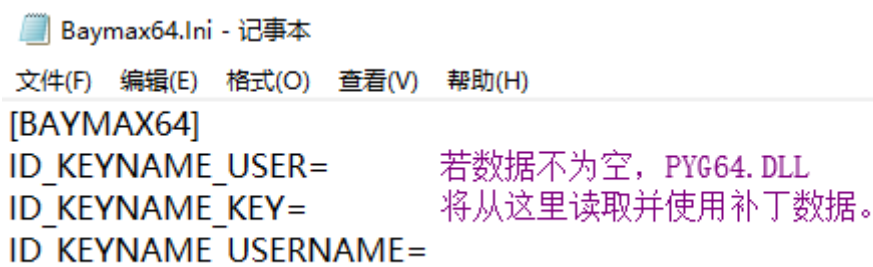
当勾选“补丁数据从文本读取的标号”后，破解模块可以通过配置文件读取用户自定义数据来完成补丁操作。



若补丁条目有勾选“补丁数据从文本读取的标号”，补丁程序除了在目标文件夹释放劫持和破解模块，还将释放 Baymax.ini 或 Baymax64.ini 文件（格式见下图），当标号项对应的数据不为空时，补丁模块将读取标号对应的数据并替换原补丁数据进行补丁。例如，软件关于信息的补丁项中若勾选该选项，用户则可自定义内容进行显示。

注意：用户填写的补丁数据须和补丁条目中值类型一致，例如补丁条目的值类型为 HEX 数值，用户输入的数据将按 HEX 数据进行解析和补丁。

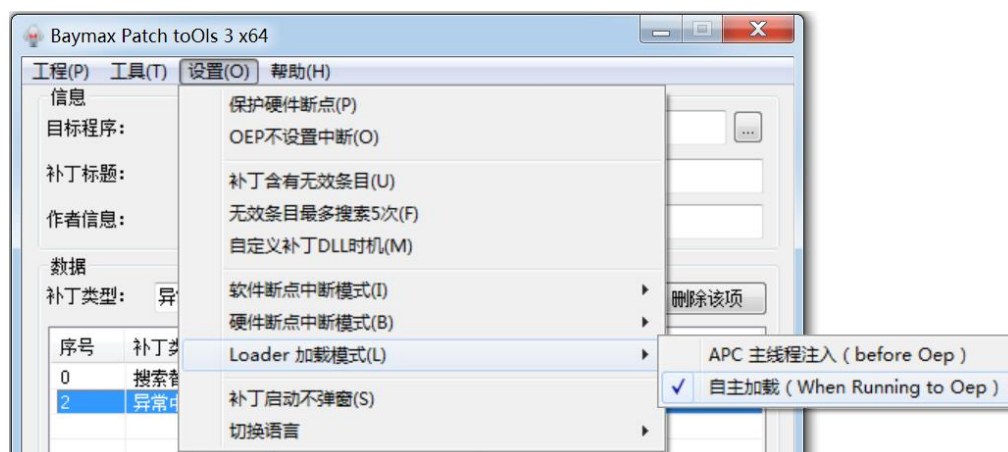




## 五. 生成补丁和保存补丁工程

### (一) 菜单设置项说明

Baymax 菜单设置项中提供了一些附加功能：



1. 保护硬件断点：某些壳会抹掉硬件断点，若分析 log 文件，发现 DrX 寄存器全部为 0，则可能设置的硬件断点被清空，此时可勾选该选项。目前 x64 中该功能还有待完善。
2. OEP 不设置中断：Baymax 的补丁默认会在 OEP 处设置 0xCC，某些壳如 Obsidium 会检测代码段完整性，此时需勾选该选项不进行设置。
3. 补丁含有无效条目：为提高启动性能，Baymax 内部进行了优化，遇到无效条目则认为解码未完成不再执行后续补丁条目。若



补丁为兼容新老版本包含无效条目，需勾选该选项。

4. 无效条目最多搜索 5 次：若包含无效条目并设置了 HOOK，每次执行 API 时都将尝试补丁，勾选该选项之后若其他补丁已执行，无效条目在执行 5 次后将标记为忽略，后续不再执行该条目。
5. 自定义补丁 DLL 时机：若补丁地址位于 DLL 模块，破解模块将默认开启监控模块加载的功能，模块加载时判定其是否为目标模块并执行补丁功能。某些大型程序启动时加载众多的 DLL 模块，由于监控模块加载会造成一定的性能损耗，影响启动速度，所以用户可勾选该选项，取消对模块加载的监控，自己把控 DLL 的补丁时机，以提高软件启动速度。
6. 软件断点中断模式：
  - 1) 模拟单步模式，x86 v2.9.5 和 x64 v2.5 之前版本 INT3 所采用的中断模式，通过解析并设置 NEXT\_IP 来模拟单步执行，一个补丁条目记录并处理一条补丁数据。
  - 2) 模拟单步模式 II，基于模式 I，函数返回地址将创建新节点进行处理。
  - 3) 模仿 HOOK 模式，支持多线程。一种全新的 INT3 中断模式，类似 HOOK 模式，触发 CC 中断时，通过异常机制，跳到新空间执行原地址的汇编指令，来实现多线程机制，推荐使用本方案。
7. 硬件断点中断模式：
  - 1) 默认设置当前线程（补丁模块被加载的线程，一般为主线

程), 非主线程可通过该线程补丁地址执行前所调用的 API 进行 HOOK, 实现对该线程设置硬件断点, 各线程拥有各自的硬件断点处理机制, 互不影响, 推荐使用本方案。

- 2) 默认设置所有线程 (通过 HOOK 对所有线程设置硬件断点), 通过 Hook API, 对新创建的线程设置循环硬件断点。注: 若新线程不会执行当前所设置的 4 个中断地址, 将没有机会设置和执行硬件列表后续的断点。

## 8. Loader 加载模式:

- 1) APC 主线程注入 (Before OEP), 通过 APC 在执行到 OEP 之前将补丁模块注入到进程。
- 2) 自主加载 (When Running to OEP), 当执行到 OEP 时, 通过 ShellCode 直接加载补丁模块。

9. 补丁启动不弹框: Baymax 补丁启动时默认展示 LOGO 弹框, 以方便用户感知补丁已正常加载工作, 若不需要展示, 可勾选此选项。

## 10. 切换语言

用户可切换所需语言, 如果您希望支持其他语言, 可将语言文件发送给我 (email or github)。

## (二) 创建劫持、注入补丁

Baymax 可以创建发布版和调试版补丁。主界面“创建补丁”按钮生成发布版劫持补丁, 菜单项中可创建调试版和注入补丁。

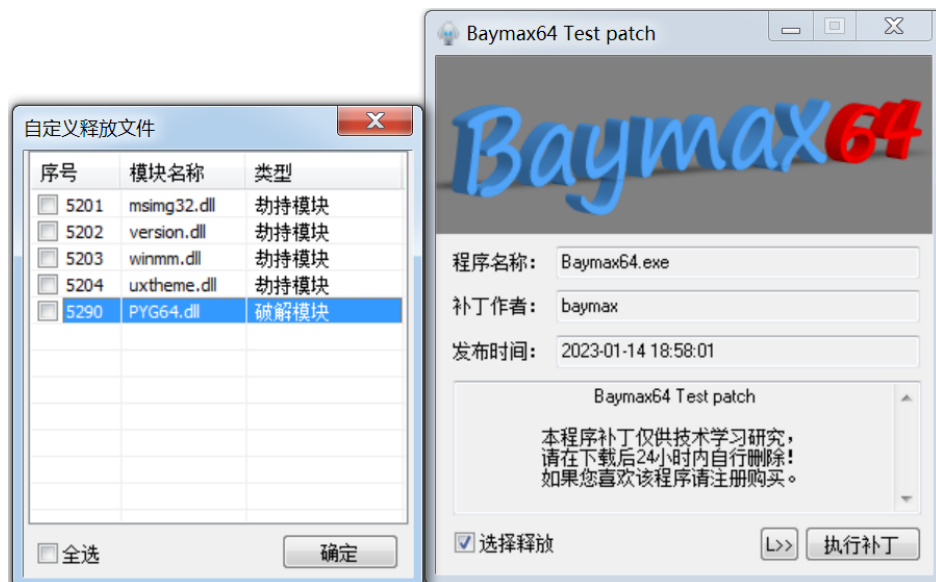
劫持补丁将劫持模块和破解模块释放至进程文件夹，通过劫持模块加载破解模块执行补丁。注入补丁启动时会将破解模块释放至进程文件夹，通过 APC 机制让进程实现加载并执行补丁。

Baymax 所创建的补丁会将已存在的语言资源保存至补丁中，补丁运行时将根据运行系统选择对应的语言（若缺失所对应的语言将展示英文），用户也可以在补丁菜单中（界面的“L>>”按钮）选择语言。

### （三）创建调试版补丁

为了便于了解补丁的执行情况和排查问题，Baymax 支持生成调试版补丁，调试版补丁和正式版补丁的区别在于：调试版补丁用户数据不进行加密，且生成记录所有补丁条目执行情况的.log 文件。

### （四）劫持模块和破解模块



Baymax 的补丁数据分两种：劫持模块和破解模块。由于不同平台，或不同的系统环境，不同的进程可加载的劫持模块是未知的。所

以 Baymax 的补丁释放工具默认采用了智能识别方式来释放劫持模块。若识别超时未找到可加载的劫持模块，补丁工具将询问是否启用注入模式启动进程，运行补丁方案。

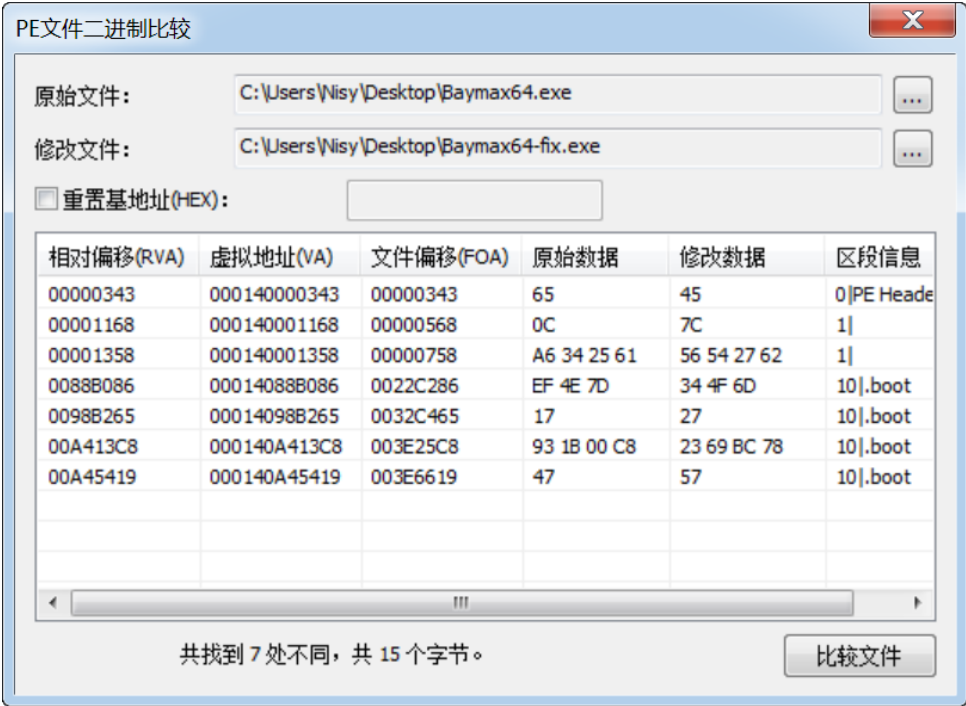
未找到可加载的劫持模块有这样两种情况：可能是补丁内的劫持模块均不能自动加载，此时你可以自己制作其他可加载的劫持模块，或者在其他加载模块中增加破解模块的导出函数进行加载，或者生成注入补丁；也可能是软件启动时未直接加载，某些大型程序在启动过程中可能会加载劫持模块，此时推荐可手动释放所有劫持模块后，运行程序通过 process explorer 等工具来查看进程是否加载了劫持模块，或通过判定程序启动是否有展示 Baymax 特有的 LOGO 弹框判定破解模块是否正常加载。

## （五）保存和打开补丁工程

Baymax 可将补丁工程保存为 \*.bpt 工程文件，支持打开 bpt 文件创建补丁。若不了解 bpt 数据格式，请勿手动编辑修改。另外 bpt 文件中包含了版本信息，当包含新版功能时，则可能在低版本的 Baymax 中创建补丁失败。新版本支持向下兼容，推荐使用最新版。

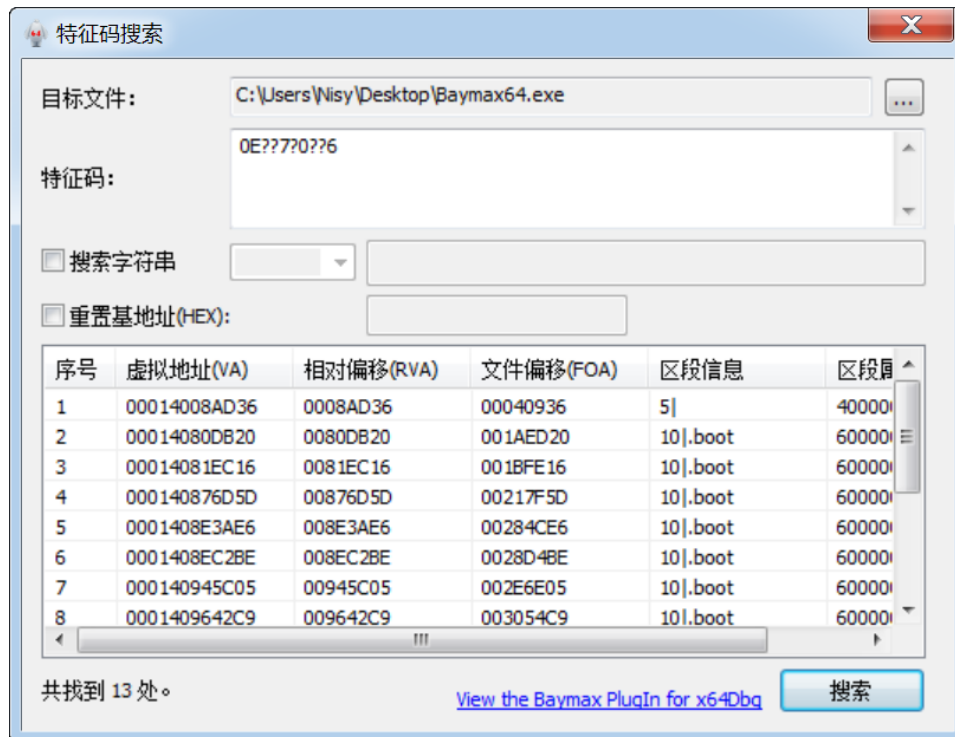
六. 插件介绍

(一) PE 文件二进制比较



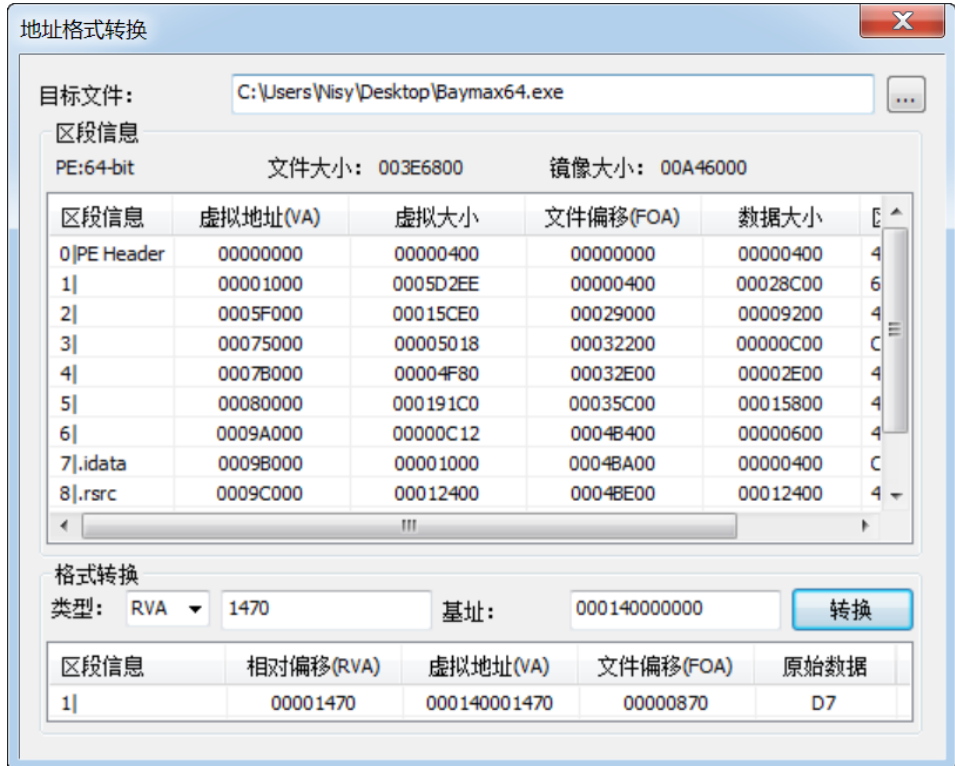
支持 PE 文件二进制比对，参与比对的文件要求大小一致。若在偏移补丁界面中通过“比较文件”按钮唤出该功能，可以点击“添加选中”将勾选补丁数据一次性添加。

## （二）特征码搜索工具



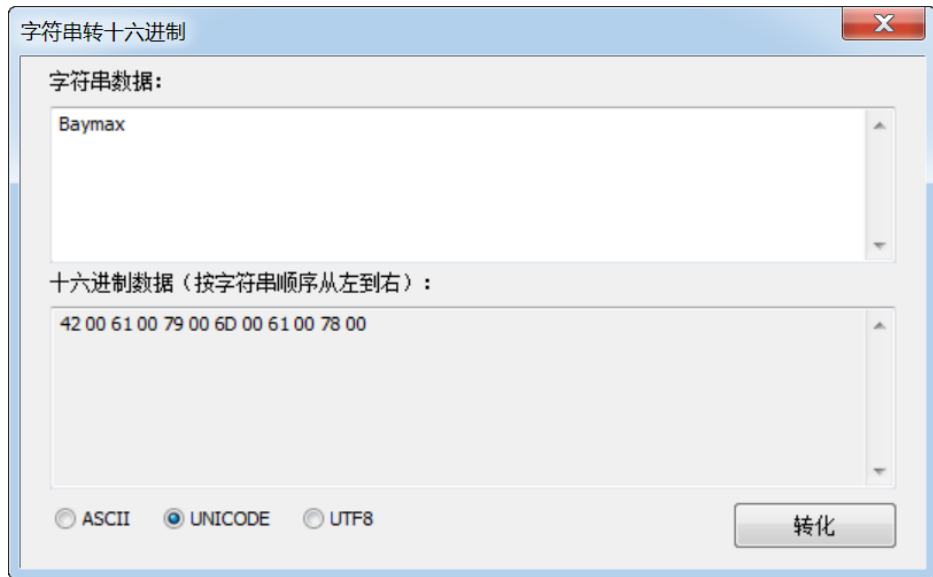
特征码搜索工具。目前只支持对 PE 文件进行检索，支持重置基址，并列出特征码对应的 VA、RVA、FOA。

### （三）地址格式转换工具



解析 PE 区段信息，支持对 RVA、VA、FOA 格式间的转换。

### （四）字符串转十六进制工具



我们在填写补丁数据时，有时需将字符串转换为 HEX 数值，所以

提供了这款小工具方便大家进行格式转换，HEX 数据按内存低地址到高地址输出。

## 七. 案例及使用技巧

### （一）通过 HOOK 实现对其他线程设置硬件断点

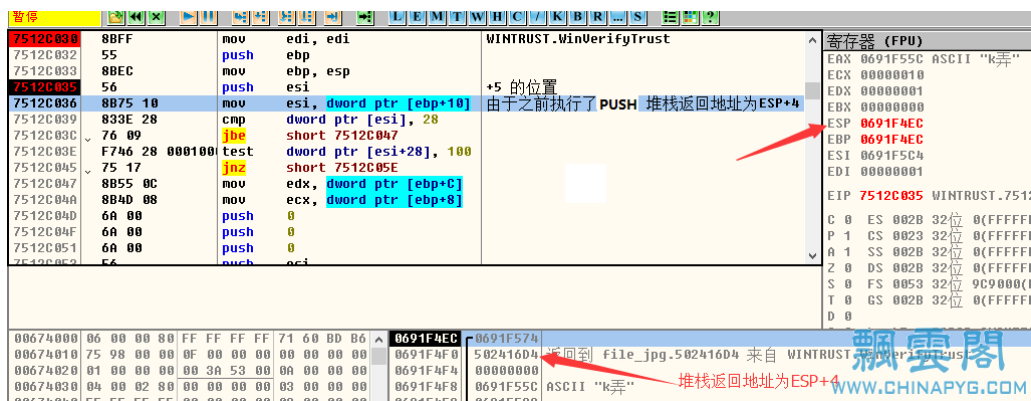
硬件断点有线程相关性，破解模块执行“断点设置函数”只对当前线程设置硬断，一般为主线程。其他线程执行到补丁地址时，由于未设置硬件不会执行补丁操作。Baymax 目前没有提供对所有线程设置硬断的选项，变通方案是 HOOK API，每执行到被 HOOK 的 API 函数时会再次调用“断点设置函数”，并对当前线程设置硬断。

### （二）通过对函数 HOOK 实现固定参数或返回值

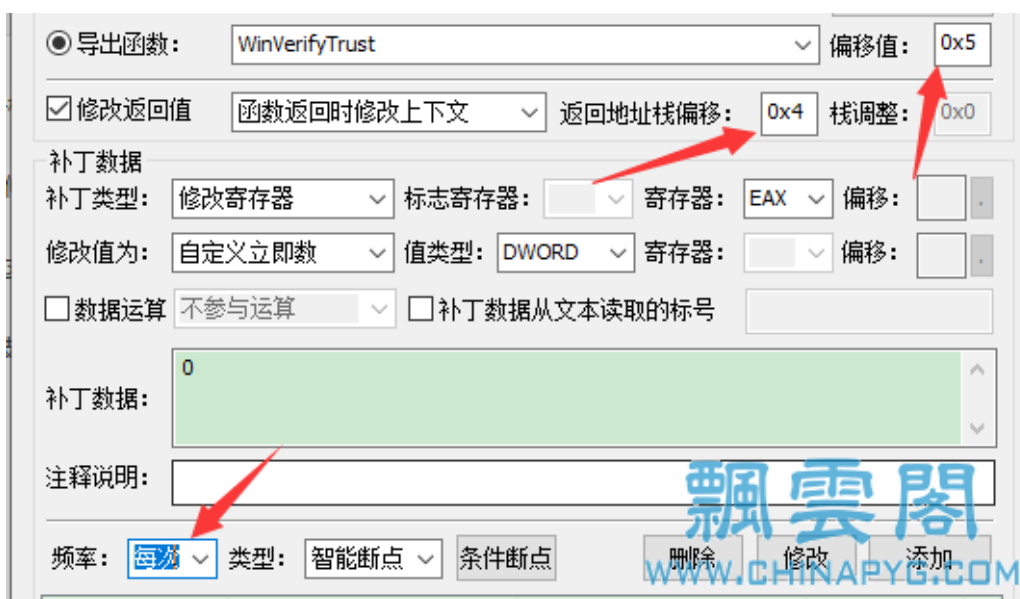
Baymax 提供的“修改函数返回值”功能有一定的局限性，若采用 INT3 方案，多线程同时调用会出现问题；若采用硬件断点，无法默认对所有线程设置断点。那依靠现有功能可否实现多线程频繁调用某函数时固定其返回值吗？答案是可以的，我们以 wintrust.dll 中导出函数 WinVerifyTrust 为例进行讲解。

首先 HOOK 该函数，这样就可以保证调用该函数的所有线程都会调用“断点设置函数”来设置硬断。然后在函数+5 的地址设置硬件断点。





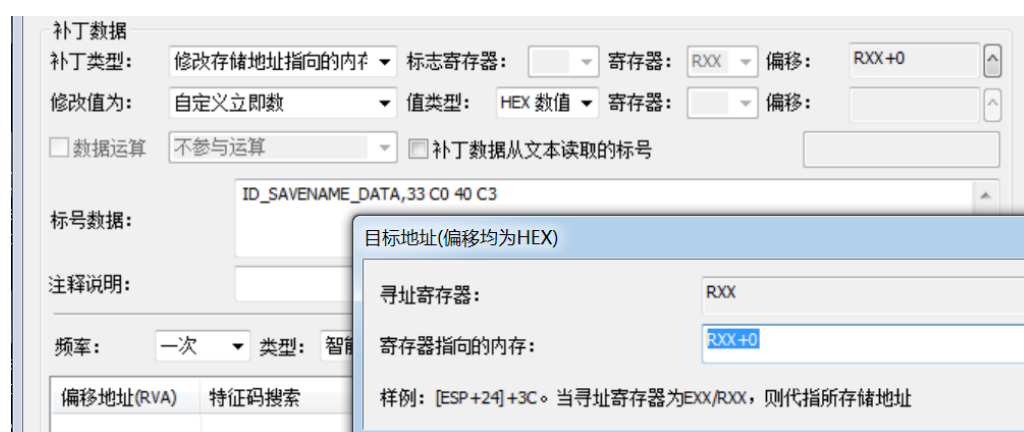
设置偏移+5 的原因是 Baymax 的 HOOK 库会将函数头修改为 JMP XXXXXXXX 指令，并在执行完内部流程后再跳回原函数被覆盖数据的下一条指令，该函数被覆盖的指令正好 5 个字节，若覆盖的指令超过 5 个字节，选择覆盖数据的下一条指令偏移值即可。勾选“修改返回值”，偏移+5 处函数的返回地址位于[ESP+4]，所以我们设置“返回地址栈偏移”为 0x4。



设置如上图，这样间接的实现了通过 HOOK 函数完成对调用线程设置硬断，从而实现固定函数返回值的功能。

### （三）通过存储数据来补丁堆空间

有些程序会将关键代码 map 到内存（堆空间）中去执行，我们可以在程序申请内存空间后保存该地址，待进程往堆空间填充完数据后，并在堆空间代码执行前的任意地址设置断点，使用存储地址+偏移来定位关键地址。



我们选择“修改存储地址指向的内存”时，发现寄存器一栏为 EXX/RXX，并且还可以设置偏移。

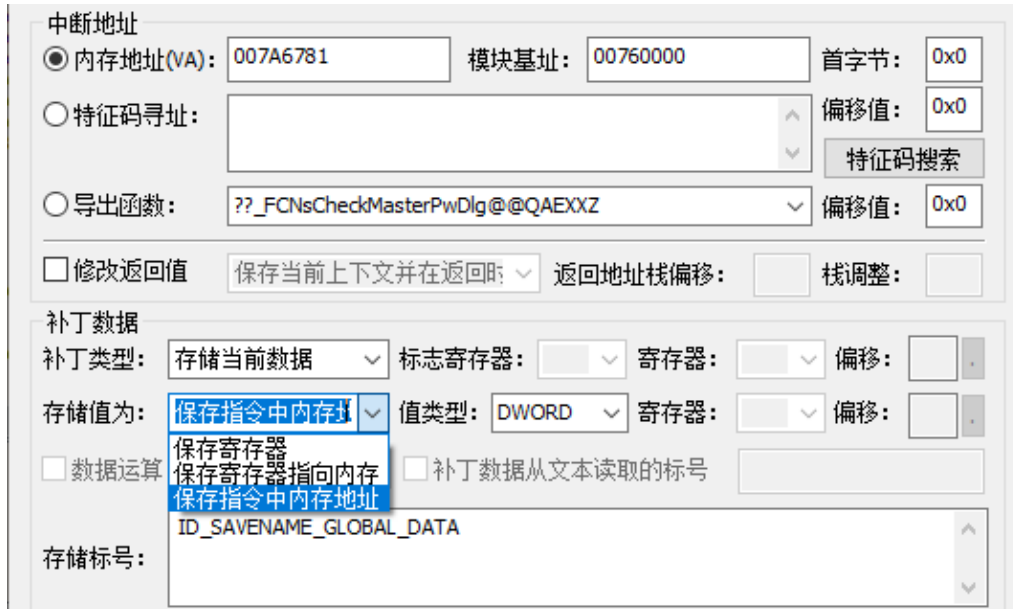
Bayamx 支持对我们保存的地址进行二次寻址，支持  $[EXX+N]+N1$  这样的寻址操作，这样我们保存类对象后，就可以方便的修改其成员的数值。例如我们保存的地址为 10000000，而我们需要修改的地址为 10000008，我们设置偏移为 EXX+8 即可（x64 位 RXX+8）。

### （四）修改指令中的全局变量值

很多程序都会将注册状态存储于数据段的全局变量，并通过 cmp 指令来判定其状态：

007A677E	880C10	mov byte ptr ds:[eax+edx],cl
007A6781	83D0 08E88100 00	cmp dword ptr ds:[81E808],0
007A6788	75 56	jne xshell.7A67E0
007A678A	B9 90000000	mov ecx,90
007A678E	EB 08	jmp ds:[ecx]

如 “007A6781 |cmp dword ptr ds:[0081E8D8],0” 指令中，我们需将 0081E8D8 的数值修改为 1，应该如何操作呢？Baymax 提供了该机制，其方法为先存储，后修改。



中断地址

☒ 内存地址(VA): 007A6781 模块基址: 00760000 首字节: 0x0

☐ 特征码寻址: 偏移值: 0x0 特征码搜索

☐ 导出函数: ??\_FCNsCheckMasterPwDlg@@QAEXXZ 偏移值: 0x0

☐ 修改返回值 保存当前上下文并在返回时 返回地址栈偏移: 栈调整:

补丁数据

补丁类型: 存储当前数据 标志寄存器: 寄存器: 偏移:

存储值为: 保存指令中内存地址 值类型: DWORD 寄存器: 偏移:

☐ 数据运算 保存寄存器 保存寄存器指向内存 保存指令中内存地址 补丁数据从文本读取的标号

存储标号: ID\_SAVENAME\_GLOBAL\_DATA

第一步：先存储。

设置中断地址，然后在补丁数据中如下设置：

补丁类型选择 “存储当前数据”；

存储值为 “保存指令中的内存地址”；

值类型为 “DWORD”（x64 中一般为 QWORD）；

存储标号中填写一个以该地址命名的唯一标号。

中断地址

☒ 内存地址(VA): 007A6781 模块基址: 00760000 首字节: 0

☐ 特征码寻址: 偏移值: 0x0 特征码搜索

☐ 导出函数: ??\_FCNsCheckMasterPwDlg@@QAEXXZ 偏移值: 0x0

☐ 修改返回值 保存当前上下文并在返回时 返回地址栈偏移: 栈调整:

补丁数据

补丁类型: 修改存数指向的内存 标志寄存器: 寄存器: EXX 偏移: 0x0

补丁数据: 自定义立即数 值类型: DWORD 寄存器: 偏移:

☐ 数据运算 不参与运算 ☐ 补丁数据从文本读取的标号

标号数据: ID\_SAVENAME\_GLOBAL\_DATA, 1

第二步，再修改。仍需设置中断地址（可以和存储地址一致），在补丁数据中如下设置：

补丁类型选择“修改存数指向的内存”；

补丁数据可选择“自定义”；

值类型“DWORD”；标号数据的格式为“标号名称，补丁数值”，此处我们应填写之前输入的标号名称和修改数值“ID\_SAVENAME\_GLOBAL\_DATA, 1”。

通过以上两步操作，我们就实现了对该地址（全局变量）的修改。

## （五）通过 HOOK 固定硬盘信息

有些程序的机器码由硬盘序列号运算生成，我们是否可以在不修改程序的情况下通过 Baymax 来固定硬盘序列号呢？

调用 DeviceIoControl 函数可以获取硬件信息（函数声明见下图），当参数 dwIoControlCode = 0x0007C088(SMART\_RCV\_DRIVE\_DATA) 时，函数返回后 lpOutBuffer 接收的数据中包含硬盘序列号信息。

```

C++
复制

BOOL DeviceIoControl(
    HANDLE      hDevice,
    DWORD       dwIoControlCode,
    LPVOID      lpInBuffer,
    DWORD       nInBufferSize,
    LPVOID      lpOutBuffer,
    DWORD       nOutBufferSize,
    LPDWORD     lpBytesReturned,
    LPOVERLAPPED lpOverlapped
);

```

我们创建中断补丁项，在补丁模块处勾选“目标为其他模块”，选择系统模块“\Windows\system32\kernel32.dll”，补丁地址从导出函数中选择“DeviceIoControl”函数。勾选“修改返回值”并选择“保存当前上下文并在返回时修改”模式，调试器在函数地址中断后栈情况如下图所示：

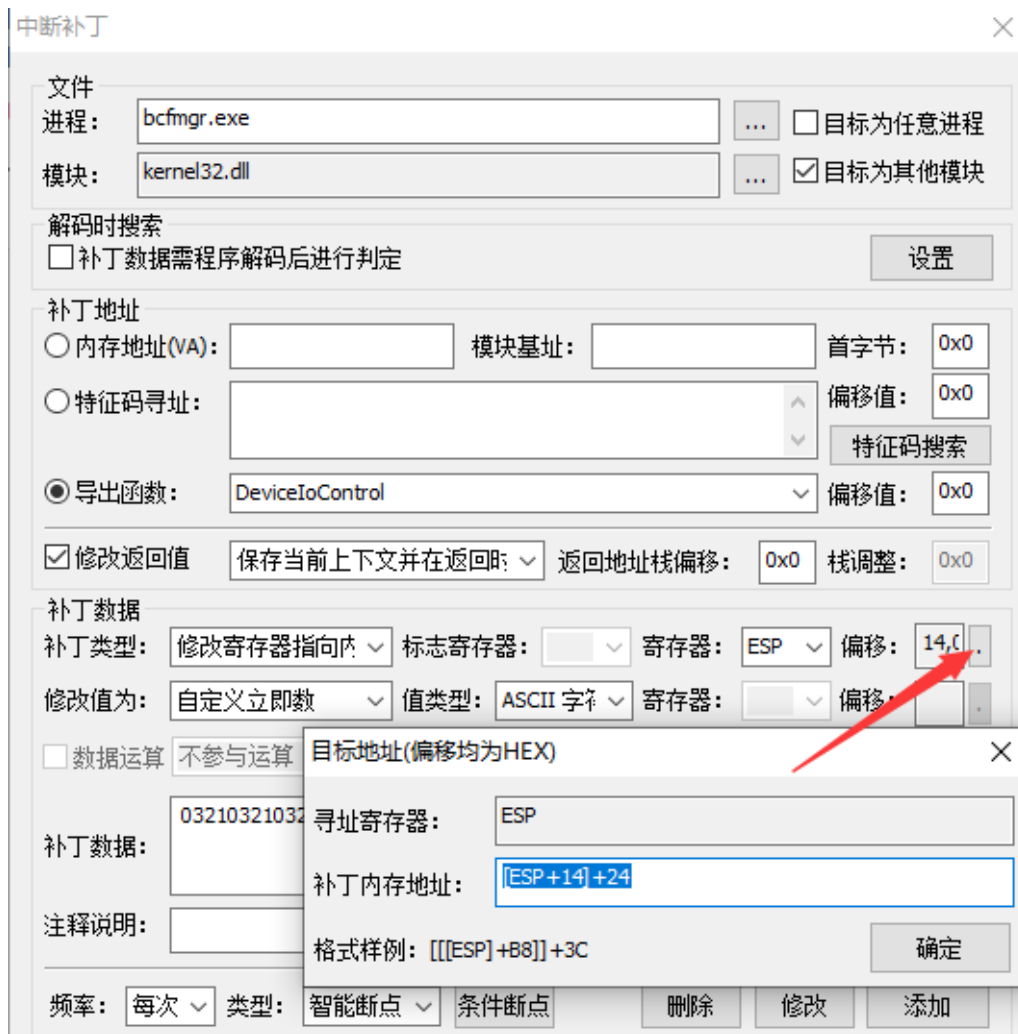
```

$ ==> > 0092DD88 GetPcInf.0092DD88
$+4   > 000000E4
$+8   > 0007C088
$+C   > 0025E9D8
$+10  > 00000021
$+14  > 0025E7C0 // 参数 lpOutBuffer
$+18  > 00000210
$+1C  > 0025EC40
$+20  > 00000000
$+24  > 0025EE78

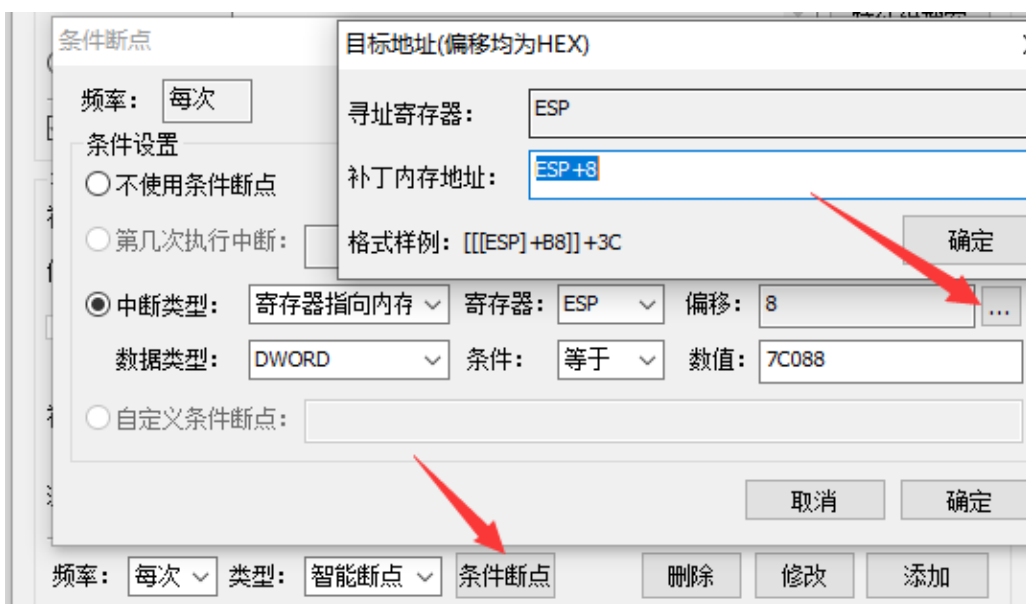
0025E7C0 00 02 00 00 00 00 01 00 00 A0 EC 00 00 00 00 00
0025E7D0 40 00 FF 3F 37 C8 10 00 00 00 00 00 3F 00 00 00
0025E7E0 00 00 00 00 30 33 32 31 30 33 32 31 30 33 32 31 ...032103210321 // 偏移0x24
0025E7F0 30 33 32 31 20 20 20 20 00 00 F0 4E 00 00 41 53 0321 ..

```

参数保存在[ESP+14]，序列号位于 lpOutBuffer 偏移 0x24 位置。我们设置补丁类型为“修改寄存器指向内存”，寄存器设定 ESP，补丁内存地址为[ESP+14]+24。



勾选“条件断点”按钮，条件为  $[ESP+8]==7C088$ 。注意，在“补丁内存地址”栏输入的永远是数据地址，即  $ESP+8$ 。设置如下：



创建补丁，运行后我们发现程序获取到的硬盘序列号已变成我们自定义的 ASCII 字符串。我们在不修改程序任意字节的情况下，固定了硬盘序列号。

## 八. 交流反馈

### （一）如何分析补丁失败原因

#### 1. 请先关闭调试器

破解模块有反调试机制，若破解模块未加载或启动时未看到 LOGO 弹框，请**关闭调试器**后再进行测试。如果用调试器来动态分析破解模块，可能会导致补丁功能异常。

#### 2. 分析调试补丁的 LOG

若补丁加载后未达到预期效果，请从菜单上选择生成调试劫持补丁或调试注入补丁，程序运行后将在进程文件夹下创建 xxx\_baymax.log 文件（若程序位于系统盘，需管理员权限运行方可创建.log 文件），32 位的进程也可打开 DbgView 工具来接收补丁输出的调试信息进行查看。我们可以通过输出日志来分析补丁执行情况找到失败原因。若无从入手也可加到工具群反馈交流，反馈时可附上 log 文件，log 中包含补丁条目和补丁地址，可修改掉关键数据后再提供，不影响问题分析。

```

1 2019-07-25 22:34:39:931: [2128][BAYMAX64]: PYG.DLL ver: 3.0.1.1025 模块加载 ↓
2 2019-07-25 22:34:39:931: [2128][BAYMAX64]: Process Attach: C:\User: [redacted]esktop\Debug\ [redacted]示DiskT
3 2019-07-25 22:34:40:117: [2128][BAYMAX64]: Not Find Baymax IniFile ↓
4 2019-07-25 22:34:40:118: [2128][BAYMAX64]: Proc DiskTest.exe Module DiskTest.exe Name DiskTest.exe ↓
5 2019-07-25 22:34:40:118: [2128][BAYMAX64]: 补丁需要进行HOOK处理: DiskTest.exe ↓
6 2019-07-25 22:34:40:119: [2128][BAYMAX64]: 增加 HOOK 2 方案 user32.dll ==> CreateWindowExW ↓
7 2019-07-25 22:34:40:119: [2128][BAYMAX64]: 初始化完成 ... ↓
8 2019-07-25 22:34:40:119: [2128][BAYMAX64]: Add Hook: Type 2 user32.dll ==> CreateWindowExW ↓
9 2019-07-25 22:34:40:119: [2128][BAYMAX64]: End StartHook() ↓
10 2019-07-25 22:34:40:119: [2128][BAYMAX64]: 补丁设置初始化完成, 若有HOOK或下断点操作, 将会在下方进行打印输出。
11 2019-07-25 22:34:40:252: [2128][BAYMAX64]: Proc DiskTest.exe Module DiskTest.exe Name DiskTest.exe ↓
12 2019-07-25 22:34:40:252: [2128][BAYMAX64]: HOOK函数, 执行补丁数据。 ↓
13 2019-07-25 22:34:40:253: [2128][BAYMAX64]: 设置断点补丁条目 ↓
14 2019-07-25 22:34:40:253: [2128][BAYMAX64]: 非 NS_TYPE_SETVRABREAK 类型 0 ↓
15 2019-07-25 22:34:40:253: [2128][BAYMAX64]: 断点补丁地址 0 [redacted] 补丁数据 RSP, [redacted]V:2,R:1,B:102,T:1,C:C
16 2019-07-25 22:34:40:253: [2128][BAYMAX64]: 设置断点 ↓
17 2019-07-25 22:34:40:254: [2128][BAYMAX64]: 解析异常断点数据成功 ThreadId: 2496 ↓
18 2019-07-25 22:34:40:254: [2128][BAYMAX64]: 设置INT3断点 ↓

```

### 3. 劫持模块加载时机过晚

有些进程，由于劫持模块加载过晚，导致破解模块加载时补丁地址已执行过，从而导致补丁无效。该情况需通过设置调试器为“在模块加载时中断”，动态调试当补丁地址中断时检查破解模块是否加载。

#### （二）加入官方交流反馈群

Baymax 官方 Q 群：112823588

#### （三）工具下载

云盘下载：<http://pan.baidu.com/s/1pLUuBEj> 密码：5x8n

GitHub：<https://github.com/sicaril/Baymax-Patch-toOls>

#### （四）更新日志

详见：<https://www.chinapyg.com/thread-83083-1-1.html>